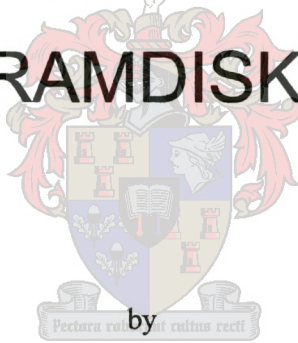


A Second Generation SUNSAT RAMDISK



A.N. Rust

*Thesis presented in partial fulfilment of the requirements for the degree of Master of
Science in Engineering at The University of Stellenbosch.*

October 2000

Supervisor: Prof. P.J. Bakkes

DECLARATION

I declare that the contents of this thesis is original and my own work unless otherwise stated and that it has not, to my knowledge, been published in part or as a whole at any other university in order to obtain a degree.

A.N. Rust

Date

SYNOPSIS

The SUNSAT RAMDISK was studied, and tests performed to assess its flight readiness. Errors were detected, and modifications had to be made to ensure proper operation. SUNSAT was subsequently launched, and to date the RAMDISK is still functioning correctly. The flight readiness testing of the RAMDISK was considered a pre-study to designing a second generation RAMDISK.

A conceptual mass memory storage device support structure was designed. This second generation RAMDISK, or memory drive is intended to be used on a second generation SUNSAT, or SUNSAT 2. The design is targeted for implementation in fields programmable gate arrays (FPGA's) and was realised using VHDL. This hardware description language is an accepted standard, and can be implemented in a number of different programmable logic devices; both SRAM and fuse-link based.

Simulations were performed to verify the functionality of the design, and to determine whether the data transfer specifications could be met using programmable logic devices.

A modular design methodology was followed. The memory drive was designed so that any type and amount of memory can be added to the drive without a major design change.

The simulations indicated that a data capturing speed of 130 Mbits/s could be maintained.

OPSOMMING

Die SUNSAT massa geheue module is bestudeer, en toetse is daarop uigever om die vlug gereedheid te bepaal. Foute is gevind, en veranderinge moes aangebring word om korrekte werking te verseker. SUNSAT is gelanseer en die geheue module werk tot op datum nog korrek. Die geheue module se vlug gereedheid toetse is beskou as 'n voor studie vir die ontwerp van 'n tweede generasie geheue module.

'n Konseptuele massa stoor toestel struktuur is ontwerp. Hierdie tweede generasie geheue module, of geheue skyf is bedoel om op 'n tweede generasie SUNSAT, of SUNSAT 2 gebruik te word. FPGA's is die teiken tegnologie vir hierdie ontwerp en VHDL is gebruik om die ontwerp te realiseer. Hierdie hardeware beskrywingstaal word as 'n standaard aanvaar en kan in verskillende herprogrammeerbare tegnologieë gebruik word.

Simulasies is gedoen om die funksionaliteit van die ontwerp te verifieer, en om te bepaal of die nodige data oordragstempo gehandhaaf kan word met herprogrammeerbare tegnologie.

'n Modulêre ontwerpfilosofie is gevolg. Die geheue skyf is ontwerp sodat dit enige tipe en hoeveelheid geheue kan ondersteun sonder om groot veranderinge aan die ontwerp te doen.

Die simulasies toon dat 'n data oordragstempo van 130 Mbits/s gehandhaaf kan word.

ACKNOWLEDGEMENTS

The author would like to thank the following:

- Prof. P. J. Bakkes (thesis supervisor) for his infinite patience.
- Liesl Rust for her love, inspiration and sitting up nights to help finish this document.

TABLE OF CONTENTS

Declaration.....	ii
Synopsis.....	iii
Opsomming.....	iv
Acknowledgements	v
Table Of Contents.....	vi
List of Figures and Tables	xi
List of Abbreviations and Acronyms	xiii
 Chapter 1	 1
Introduction.....	1
1.1 SUNSAT	1
1.2 Project scope	2
1.2.1 <i>Satellite memory requirements</i>	<i>2</i>
1.2.2 <i>Reconfigurable Logic</i>	<i>3</i>
1.2.2.1 <i>FPGA's.....</i>	<i>3</i>
1.2.3 <i>Simulation.....</i>	<i>3</i>
1.2.3.1 <i>VHDL</i>	<i>4</i>
 Chapter 2	 5
Sunsat RAMDISK	5
2.1 Introduction	5
2.2 SUNSAT RAMDISK.....	6
2.2.1 <i>Overview.....</i>	<i>6</i>
2.2.2 <i>64 Mbytes SRAM.....</i>	<i>8</i>
2.2.3 <i>Clock Sources and Pulse Generators.....</i>	<i>8</i>
2.2.4 <i>Analog to Digital Converters.....</i>	<i>9</i>
2.2.5 <i>Multiplexers and Data Routers.....</i>	<i>9</i>
2.2.6 <i>Telemetry Latches</i>	<i>10</i>
2.2.7 <i>Telecommand Interface</i>	<i>10</i>
2.2.8 <i>Line and Pixel Counters</i>	<i>10</i>
2.2.9 <i>L-Band Uplink Serial to Parallel Converter</i>	<i>11</i>
2.2.10 <i>S-Band downlink Serialiser</i>	<i>11</i>
2.2.11 <i>Inline FIFO.....</i>	<i>12</i>
2.2.12 <i>Failed Daughter board management PAL</i>	<i>13</i>
2.2.13 <i>Compression FPGA.....</i>	<i>13</i>
2.2.14 <i>Power Control.....</i>	<i>14</i>
2.3 Faults Encountered and Solutions Found.....	15
2.3.1 <i>Debug and Testing Methods</i>	<i>16</i>
2.3.1.1 <i>Analysis of Schematics.....</i>	<i>17</i>
2.3.1.1.1 <i>Modification Considerations</i>	<i>18</i>
2.3.1.1.2 <i>Modification Process</i>	<i>19</i>
2.3.1.1.2.1 <i>Design of the modification</i>	<i>19</i>
2.3.1.1.2.2 <i>Qualification of the design</i>	<i>19</i>
2.3.1.1.2.3 <i>Discussion of the design.....</i>	<i>20</i>
2.3.1.1.2.4 <i>Implementation on the engineering model RAMDISK</i>	<i>20</i>
2.3.1.1.2.5 <i>Final approval.....</i>	<i>21</i>

2.3.1.1.2.6	Implementation on the flight model RAMDISK	21
2.3.1.1.2.7	Qualification testing	21
2.3.1.2	Software Development	22
2.3.1.3	Measurement and Testing	22
2.3.1.4	Comparison of results	22
2.3.2	<i>Final Qualification</i>	23
2.3.3	<i>SUNSAT RAMDISK Design Errors</i>	23
2.3.3.1	L-Band Uplink Serial to Parallel Converter	23
2.3.3.2	Clock Generation Circuit	24
2.3.3.3	CCD Clocks	25
2.3.3.4	Telecommand Interface	26
2.3.3.5	Reset Circuitry	27
2.3.3.6	Line and Pixel Counters	28
2.3.3.7	FIFO	29
2.3.3.8	Source Switching Problem	30
2.3.3.9	PAL Program	30
2.3.3.10	Black Reference	30
2.3.3.11	S-Band Serialiser	31
2.3.3.12	Analog to Digital Converters	33
2.4	RAMDISK Software	34
2.4.1	<i>Software Functions</i>	35
2.5	Conclusions and Lessons Learned	38
Chapter 3	39
Memory Drive	39
3.1 Introduction	39
3.2 Design Discussion	40
3.2.1	<i>Speed</i>	40
3.2.1.1.1	Speed Requirement	41
3.2.2	<i>Memory Size</i>	41
3.2.3	<i>Reliability</i>	42
3.2.4	<i>Power Consumption</i>	43
3.2.5	<i>Physical Size</i>	44
3.2.6	<i>Cost</i>	44
3.2.7	<i>Redundancy</i>	45
3.2.8	<i>Complexity</i>	45
3.3 Design Overview	46
3.3.1	<i>Central Data Router</i>	49
3.3.2	<i>Generic Memory Interface</i>	49
3.3.3	<i>Command Register</i>	50
3.3.4	<i>Memory Interface Power Control</i>	51
3.3.5	<i>Serial To Parallel Converter</i>	51
3.3.6	<i>Parallel To Serial Converter</i>	51
Chapter 4	52
Detail Design	52
4.1 Introduction	52
4.2 Design Structure	53
4.3 Design Tradeoffs and Choices	53
4.3.1	<i>Drive Structure</i>	53

4.3.1.1	Data Bus Structure.....	54
4.3.1.1.1	Data Bus Width	54
4.4	Altera Implementation.....	56
4.4.1	<i>Generic Memory Interface.....</i>	<i>58</i>
4.4.1.1	GMI Selection Process	58
4.4.1.2	Address Structure	59
4.4.1.3	GMI State Machine	60
4.4.1.4	Interface to the MSI.....	60
4.4.1.5	Spanning Multiple GMI's.....	61
4.4.1.6	External Buffers	62
4.4.2	<i>Central Data Router.....</i>	<i>62</i>
4.4.2.1	Data Paths	64
4.4.2.2	Data Router State Machine	65
4.4.2.3	Power Control	67
4.4.3	<i>Command Register.....</i>	<i>67</i>
4.4.3.1	Commands	68
4.4.4	<i>Serial to Parallel Converter</i>	<i>70</i>
4.4.4.1	Circular Buffer	71
4.4.4.2	Shift Register.....	73
4.4.5	<i>Parallel to Serial Converter</i>	<i>75</i>
Chapter 5	76
Detail Simulation	76
5.1	Introduction	76
5.2	Platform.....	76
5.3	Simulations.....	77
5.3.1	<i>Data Router and Command Register Simulation.....</i>	<i>77</i>
5.3.1.1	Address Set-Up Cycle	79
5.3.1.2	OBC Write Cycle	80
5.3.2	<i>GMI Simulation.....</i>	<i>80</i>
5.3.2.1	GMI Address Set-up.....	80
5.3.2.2	GMI Write and Read Cycle	81
5.3.3	<i>Data Router with a GMI Connected Simulation.....</i>	<i>82</i>
5.3.3.1	System Wide Address Set-up Simulation.....	84
5.3.3.2	System Wide Write Cycle Simulation.....	85
5.3.3.3	System Wide Read Cycle Simulation.....	86
5.3.3.4	Power Control Simulation	87
5.3.3.5	Serial to Parallel Converter Simulation.	87
5.3.3.6	System Wide Serial To Parallel Converter Simulation	88
5.3.3.7	System Wide Parallel to Serial Converter Simulation	88
Chapter 6	90
Conclusion	90
6.1	Sunsat RAMDISK.....	90
6.2	Memory Drive	90
6.2.1	<i>Implementation Notes and Suggestions.....</i>	<i>91</i>
6.2.1.1	Different implementations settings	91
6.2.1.2	Redundancy.....	91
6.2.2	<i>Further Research</i>	<i>92</i>

References	93
Appendix A	94
Memory Drive Signal Listings	94
A.1 Introduction	94
A.2 Signal Listings	94
<i>A.2.1 Data Router and Command Register Signals</i>	<i>94</i>
A.2.1.1 Input/Output Signals.....	94
A.2.1.1.1 General I/O	94
A.2.1.1.2 Command Register I/O	94
A.2.1.1.3 Generic memory interface I/O.....	95
A.2.1.1.4 Serial to Parallel Converter I/O	95
A.2.1.1.5 Parallel to Serial Converter I/O	95
A.2.1.1.6 Power Control I/O	95
A.2.1.2 Internal Signals.....	96
A.2.1.2.1 Signals used to interface with the Parallel To Serial Converter....	96
A.2.1.2.2 Data Router State Types	96
A.2.1.2.3 Data Router Internal Signals	97
A.2.1.2.4 Command Register Static Variable and Command Table	97
A.2.1.2.5 Command Register State Types	98
A.2.1.2.6 Command Register Internal Signals	98
<i>A.2.2 Generic Memory Interface Signals</i>	<i>99</i>
A.2.2.1 Input/Output Signals.....	99
A.2.2.1.1 General I/O	99
A.2.2.1.2 Central Data Router I/O	99
A.2.2.1.3 Memory Secific Interface I/O	99
A.2.2.1.4 Testing Outputs	100
A.2.2.2 Internal Signals.....	100
A.2.2.2.1 Constants and Command Table.....	100
A.2.2.2.2 Memory Interface State Types	100
A.2.2.2.3 Memory Interface Internal Signals	101
<i>A.2.3 Power Control Signals</i>	<i>102</i>
A.2.3.1 Input/Output Signals.....	102
A.2.3.1.1 General I/O	102
A.2.3.1.2 Central Data Router I/O	102
A.2.3.1.3 Internal Signal	102
<i>A.2.4 Parallel to Serial Converter Signals</i>	<i>102</i>
<i>A.2.5 Serial to Parallel Converter Signals</i>	<i>103</i>
A.2.5.1 Shift Register Signals	103
A.2.5.2 Circular Buffer Signals	103
A.2.5.3 Input/Output Signals.....	103
A.2.5.3.1 Genral I/O.....	103
A.2.5.3.2 Central Data Router I/O	103
A.2.5.3.3 Shift Register I/O.....	104
A.2.5.4 Internal Signals.....	104
Appendix B	105
Memory Drive VHDL Code	105
B.1 Data Router And Command Register (Router.vhd).....	105
B.2 Generic Memory Interface (memInt.vhd).....	114

B.3 Power Control (PowerCtrl.vhd) 118
B.4 Circular Buffer (CircBuff.vhd) 118

LIST OF FIGURES AND TABLES

Figure 1 – SUNSAT Image of Malta	2
Figure 2 – SUNSAT RAMDISK Flight Model	5
Figure 3 – SUNSAT RAMDISK Daughter Boards and External Systems	6
Figure 4 – SUNSAT RAMDISK Overview	7
Figure 5 – SUNSAT RAMDISK Debugging Method	17
Figure 6 – L-Band Serial to Parallel Converter	24
Figure 7 – Clock Generation Circuitry	25
Figure 8 – CCD Clock Generation Circuit	26
Figure 9 – Telecommand Interface	27
Figure 10 – Line and Pixel Counters	28
Figure 11 – FIFO	29
Figure 12 – Interrupt and Black Reference Generators	31
Figure 13 – S-Band Serialiser and Clock Divisor Circuit	32
Figure 14 – S-Band Serialiser Measurements	33
Figure 15 – Analog to Digital Converters	34
Figure 16 – Memory Drive Block Diagram	48
Figure 17 – Memory Drive Implementation in Max+Plus II	57
Figure 18 – Generic Memory Interface Block Diagram	58
Figure 19 – OverFlow: Passing Control From One GMI to the Next	61
Figure 20 – Central Data Router and Command Register Block Diagram	63
Figure 21 – Data Router OBC Read Cycle	66
Figure 22 – Serial to Parallel Converter	72
Figure 23 – Clock Splitting Process	73
Figure 24 – Shift Register	74
Figure 25 – Parallel to Serial Converter	75
Figure 26 – Central Data Router and Command Register Simulation	78
Figure 27 – Router Address Set-up	80
Figure 28 – Router Write Cycle	80
Figure 29 – GMI Address Set-up Simulation	81
Figure 30 – GMI Write and Read Cycle Simulation	82
Figure 31 – Data Router with a GMI Connected Simulation	83

Figure 32 – System Wide Address Set-up Simulation84

Figure 33 – System Wide Write Cycle Simulation85

Figure 34 – System Wide Read Cycle Select GMI Simulation86

Figure 35 – System Wide Read Simulation86

Figure 36 – Power Control Simulation.....87

Figure 37 – Serial to Partallel Converter Simulation87

Figure 38 – System Wide Serial To Parallel Converter Simulation.....88

Figure 39 – System Wide Parallel to Serial Conversion Simulation89

Table 1 – Proposed Imager Parameters42

Table 2 – Command Code Table.....70

LIST OF ABBREVIATIONS AND ACRONYMS

ADCS	Attitude Determination and Control System
CAN	Controller Area Network
CCD	Charge coupled device
DAC	Digital to Analog Converter
DRAM	Dynamic RAM
FET	Field Effect Transistor
FPGA	Field Programmable Gate Array
Gbytes	Gigabytes
GDF	Graphic Definition File
GMI	Generic Memory Interface
HDL	Hardware Description Language
IC	Integrated Circuit
LSB	Least Significant Bit
Mbits	Megabits
Mbytes	Megabytes
MHz	Mega Hertz
MSB	Most Significant Bit
MSI	Memory Specific Interface
OBC	On Board Computer
SEU	Single Event Upset
LEO	Low Earth Orbit Satellite
QPSK	Quadrature Phase Shift Key
PAL	Programmable Array Logic
RAM	Random Access Memory
RF	Radio Frequency
SRAM	Static RAM
TTL	Transistor-Transistor Logic
Vcc	Digital Power Rail
VHDL	VHSIC HDL
VHSIC	Very High Speed IC
us	micro second

Chapter 1

INTRODUCTION

1.1 SUNSAT

In 1991 an ambitious project was started at the department of Electric and Electronic Engineering at the University of Stellenbosch. South Africa's first satellite was conceptualised by students and lecturers at the department whom drew up preliminary designs of SUNSAT (Stellenbosch University Satellite). This is a 60 Kg micro satellite with a high-resolution imager as its main payload.

SUNSAT was launched from the Vandenburg Airforce Base on the 11th attempt at 10h29:45 GMT on 23 February 1999. It was sent into orbit with a Delta II launch vehicle.

Payloads include NASA experiments, Radio Amateur communications, a high-resolution imager, precision attitude control, and school experiments.

This thesis in part contributed to the mass memory system aboard SUNSAT. The RAMDISK is used to store images taken with the imager, and house the OBC (on board computer) filing system. Figure 1 is an image of Malta taken by SUNSAT on the 14'th of July 1999. This image was stored on the RAMDISK before being downloaded to earth.

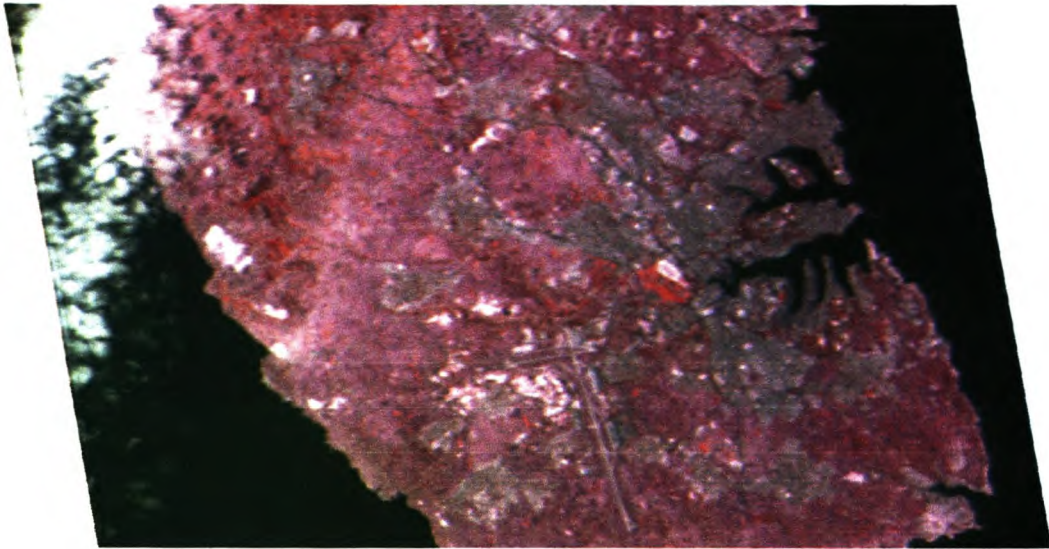


Figure 1 – SUNSAT Image of Malta

1.2 Project scope

The process of debugging and getting the SUNSAT RAMDISK ready for flight was part of this thesis and will be explored in Chapter 2.

The main aim of this project is to conceptualise, design and simulate a next generation mass memory storage device. The lessons learned from working on the SUNSAT RAMDISK were used to design this device. This device will be referred to as a memory drive.

1.2.1 Satellite memory requirements

Imaging satellites are typically in low earth orbit (LEO) to facilitate high-resolution images. This implies that a ground station can only see it for a fraction of its lifetime.

It is not plausible for a satellite to only take images when in range of a ground station, which means a storage mechanism is needed. This is normally achieved with a mass memory storage device. This device must be able to capture large amounts of data at high speed.

The high-speed requirement arises from the fact that an imager generates large amounts of data in a short space of time. This is due to the high ground speed of a LEO satellite [15].

1.2.2 Reconfigurable Logic

The SUNSAT RAMDISK was built with discrete logic devices, which made it large in physical size, and difficult to debug.

With today's advances in the programmable logic field all the functionality needed can be programmed into one device. A class of these devices suitable for a large logic design is Field Programmable Gate Arrays or FPGA's.

1.2.2.1 FPGA's

FPGA's supply the three elements sought by digital designers: storage elements, logic and wires [12]. Logic performs boolean functions and storage elements can store logic states. The wires connect everything together. This enables a designer to make an application specific circuit, which can easily be programmed into a FPGA.

The use of programmable logic devices can cut digital circuit design time dramatically, because the designer can immediately see the results of a design, and via software easily correct any faults.

FPGA architecture consists of configurable logic blocks and interconnect resources or switch matrices [6]. The logic blocks can contain simple logic gates, for a combination of flip-flops and logic gates. With these resources combinational and sequential logic functions can be designed.

Various technologies are used to implement FPGA configuration: Static RAM cells, anti-fuses, EEPROM or EPROM transistors.

1.2.3 Simulation

These programmable devices are programmed with the help of design software. The logic function can be entered on a computer, and then compiled to configure a programmable device like an FPGA.

These designs can also be simulated to verify operation before being programmed into silicon. This is a very powerful tool, as faults can be identified and corrected before hardware is touched. VHDL can be used to describe the behaviour of a FPGA.

1.2.3.1 VHDL

The United States Department of Defence, as part of its Very-High-Speed Integrated Circuit (VHSIC) program, developed VHSIC HDL or VHDL in 1982 [13]. HDL is an acronym for Hardware Description Language. The behaviour, function, inputs and outputs of a digital circuit can be described using VHDL.

VHDL has the following advantages over traditional design methodologies [3]:

- Design functionality can be verified early in the design process by using a simulator. Conceptual errors can easily be identified.
- Hardware compilers can directly translate and optimise a VHDL description to an internal gate-level equivalent.
- VHDL descriptions provide technology-independent documentation of a design and its functionality. A VHDL description is more easily read and understood than a netlist or schematic description. Since the initial VHDL design description is technology-independent, you can later reuse it to generate the design in a different technology and on a different development platform.
- VHDL is recognised as a standard HDL by the IEEE (IEEE Standard 1076, ratified in 1980) and by the United States Department of Defence (MIL-STD-454L). [13]

Chapter 2

SUNSAT RAMDISK

2.1 Introduction

The aim of the SUNSAT project was not only to launch a functional satellite into space, but also to provide a vehicle for students to obtain a Masters degree. Many of the subsystems aboard SUNSAT were worked on by more than one person as students finished their Masters degrees, and the design or implementation was passed on to another student. This was the case for the RAMDISK.

Work on the RAMDISK was considered a pre-study to conceptualising and designing a storage device for the next generation on SUNSAT satellites. This consisted of studying the present design, identifying and correcting any faults encountered, with the aim of qualifying the RAMDISK for operation aboard SUNSAT. Errors crept in due to the design being modified by a number of students and extra functionality added, especially in interfaces between structures designed by different people. This presented an interesting problem, not only must solutions be found for design errors, but different designs must also be merged.

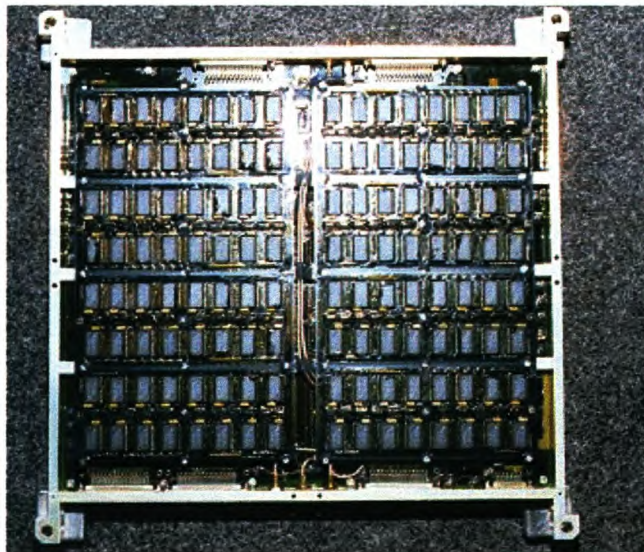


Figure 2 – SUNSAT RAMDISK Flight Model

The focus of this thesis is not on the SUNSAT RAMDISK, the process of readying it for flight is considered a pre-study. The lessons learned and experience gained will be used in designing a new storage device. Thus the following discussion will not be an exhaustive explanation of the RAMDISK, but an overview. Where solutions were implemented more detail will be provided. Please see [4] for complete design.

Figure 2 is a photograph of the SUNSAT RAMDISK flight model. The layer that can be seen consists of the daughter boards and the motherboard is hidden beneath it.

2.2 SUNSAT RAMDISK

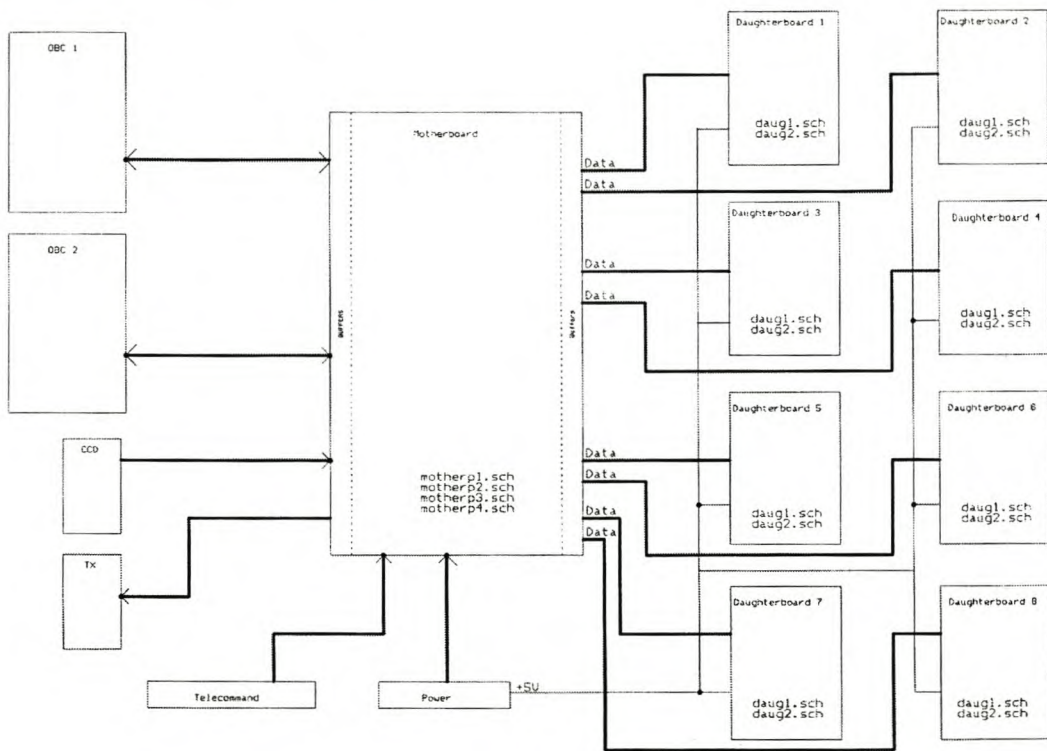


Figure 3 – SUNSAT RAMDISK Daughter Boards and External Systems

2.2.1 Overview

As already mentioned, SUNSAT's main payload is a pushbroom, three band imager with a potential resolution of 15×15 m. A pushbroom imager collects lines of data to from an image, or scans the earth's surface line by line. The three sensors (one for each band) used in the SUNSAT imager are 3490 element line CCD's. The image data from these sensors are stored, or routed through the RAMDISK.

The RAMDISK is also the main storage device aboard the satellite, and occupies an entire tray. The RAMDISK components include:

- 64 Mbytes of SRAM.
- Clock sources and pulse generators for the imager
- Three 8-bit Analog to Digital converters which sample CCD levels
- Multiplexers and data routers.
- Telemetry Latches which inserts Telemetry data into the image stream
- Telecommand Interface
- Line and pixel counters
- L-Band uplink serial to parallel converter
- S-Band downlink serialiser
- Inline FIFO for discarding every second image line
- PAL which controls on/off blocks of RAM
- Compression FPGA
- Power Control

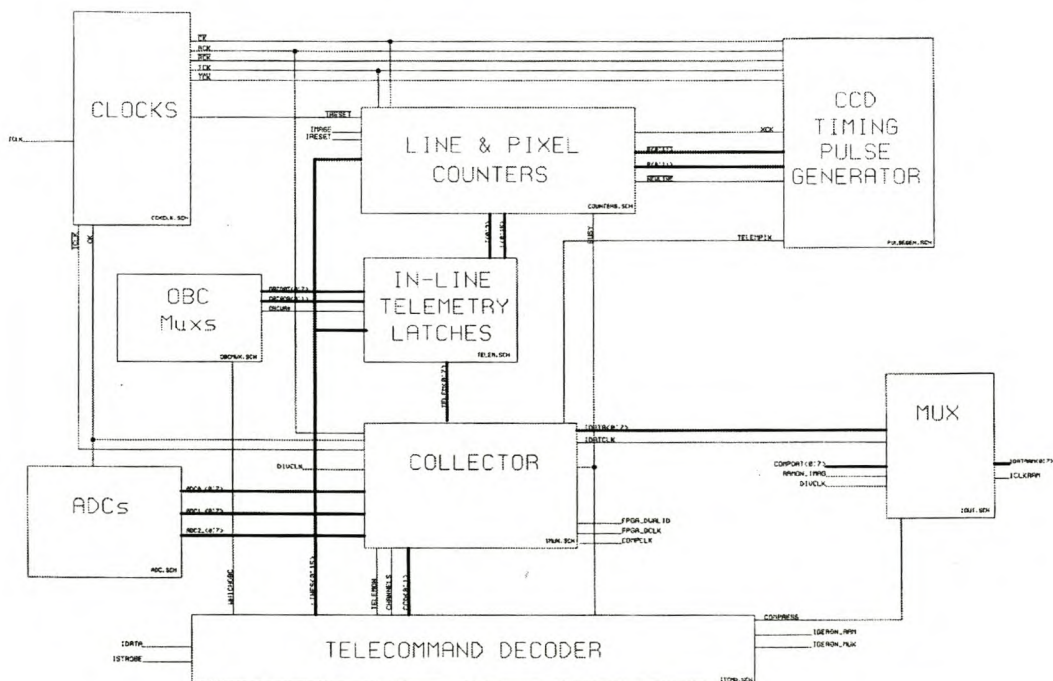


Figure 4 – SUNSAT RAMDISK Overview

2.2.2 64 Mbytes SRAM.

The 64 Mbytes are divided into eight daughter boards. Each daughter board contains sixteen 512 Kb SRAM devices to make up 8 Mbytes. Counters on the daughter boards generate the addresses. Each daughter board read or write operation then increments these and the address is then set-up for the next operation. When the end of the daughter board memory space is reached an overflow is generated which transfers control to the next daughter board.

The eight daughter boards fills up the area of one tray on the satellite, thus the configuration of the RAMDISK consist of two layers

Figure 3 illustrates how the daughter boards are connected to the motherboard. The other external support structures are also shown.

2.2.3 Clock Sources and Pulse Generators

The RAMDISK controls the imager activation and speed. These two systems are actually highly integrated. Most of the start pulses, clocks and control signals originate on the RAMDISK.

There are four clock sources, three are generated by oscillator packages and one is sourced directly from the L-band. Two are normal oscillator packages, one operating at 25 MHz and the other at 40 MHz. The remaining package is programmable, its range stretching from 360 KHz to 120 MHz. The L-band uplink has a data rate of 2 Mbits/s, thus the clock is 2 MHz.

SUNSAT's orbit is not of constant altitude. It varies between 500 and 800 Km. The lower the satellite, the faster it moves relative to the earth's surface. This necessitates that the speed at which the lines are sensed must be changed in order to keep the pixels square. See 3.2.1 for a more detailed discussion. Clock speeds vary between 33 MHz for high altitudes to 60 MHz for lower altitudes. Note that this is the speed at which the data is clocked into the RAM, and the signals that are sent to the imager are derived from this clock.

2.2.4 Analog to Digital Converters

A CCD senses the amount of light that is focused on it by the imager's lens. This light intensity is converted to a voltage level. These voltages are then shifted in a serial fashion to the RAMDISK via coaxial cable. Here it is sampled by an 8-bit analog to digital converter.

The CCD's are charged up to a voltage level, and then discharges proportionally to the intensity of the light that is focused upon it. If it is saturated the voltage level is at its lowest. Thus highest intensity light produces 0 when digitised and no light produces 255. The sampling of the CCD voltages is critical, because they have an RC, and not a step response. Sampling must only take place after the final value has been reached. This point is derived from the clocks that are sent to the imager.

There are three DAC's, one for each colour band.

2.2.5 Multiplexers and Data Routers

The RAMDISK has a large number of multiplexers. This is because data can originate from a number of places, and also be sent to a number of places. There are three main sets of multiplexers. Each set receives data from a number of sources and then selects one of these to continue to the next stage.

Data sources:

- The three DAC; this is the image data directly from the imager
- Telemetry data; information which is inserted into the image.
- L-band serial to parallel converter; L-band uplink data source
- OBC 1 & 2; data that is written to the RAMDISK from the OBC's
- RAM; data that is read from the RAM

Data destinations:

- RAM; data that is written to the RAM
- OBC 1 & 2; the OBC's can also read data from the RAMDISK
- S-band downlink; data that is sent down to earth via the S-band link

2.2.6 Telemetry Latches

The telemetry latches gather information from different sources and then inserts the data into the image. This is a very useful function because information relating to the image such as time when it was taken, position, etc. is immediately available in the image itself.

The latches are all connected to a common bus, 8-bits wide. They are either tri-stated, or transparent. When transparent, the data from that latch is placed on the telemetry bus, which is eventually included into the image using multiplexers. See 2.2.5.

There are three sets of four latches and each set has a different source. The first set includes line information: total number of lines in this image (2 bytes) and current line (2 bytes). The next set includes ADCS data (4 bytes). This may be navigational data such as pitch, roll and yaw of the satellite. The last set contains OBC data (4 bytes). This data is set via software, and can typically contain information such as time that the image was taken, or environmental data, etc.

2.2.7 Telecommand Interface

Most of the commands sent to and settings made on the RAMDISK are done via telecommands. Most of these control signals are constant and not instantaneous controls. Also many settings are made using one data line. Thus shift registers are used to set the control lines. The commands are sifted to their correct position using the data line, and using another telecommand line to serve as clock for the sift registers.

2.2.8 Line and Pixel Counters

The imager aboard SUNSAT has got very little intelligence built in. As already mentioned the RAMDISK switches the imager on and off. When images are taken remotely they are stored in memory, as apposed to being sent down to earth directly via the S-band transmitter. In this situation the imager is switched off automatically by the RAMDISK after the required amount of lines has been stored.

This is done by the line and pixel counters on the RAMDISK. The amount of lines wanted is loaded into counters (line counters) via telecommands and another set of

counters (pixel counters), which is hard coded to 3490, increments the line counters with every overflow. When the line counters overflows a pulse is created that switches the imager off and also inhibits further write operations to the memory.

The line counters are composed of four 4-bit counters. This means that a maximum of 65536 lines can be taken, which is more than sufficient as the memory storage capability is less than this. An overflow of the counters creates the stop pulse, thus to capture one line the counters must be loaded with 65534 (overflow occurs at 65535).

2.2.9 L-Band Uplink Serial to Parallel Converter

The L-band uplink has data bandwidth of 2 Mbits/s. The uplink path can be routed to the RAMDISK, and is a fast method to upload data. One application is to load an OBC boot image into memory, and then to boot from the RAMDISK.

The data and clock is sent to the RAMDISK with differential lines. These signals are received by a LTC489 package, which converts them to TTL levels. The clock is divided by eight, and a parallel clock is generated. The data is shifted into an 8-bit shift register and each 8th bit coincides with a rising edge of the parallel clock. This rising edge clock the byte into memory.

The L-band data is received serially. This means that the bytes stored in memory are not necessarily correct data bytes, bit zero of a byte might be at any position, e.g. at bit three, which means to get the proper data the whole sequence must be shifted right by three. A synchronisation sequence, or header must be appended to a block of data in order for the start byte to be known.

2.2.10 S-Band downlink Serialiser

The S-Band downlink serialiser is the high-speed download path for images. Data from the RAMDISK is serialised and sent to the QPSK (Quadrature Phase Shift Key) modulator, where it is packaged in QPSK format and ultimately transmitted down to earth. Data transfer speeds of up to 40 Mbits/s can be achieved. A UHF QPSK modulator and transmitter serve as a backup.

Images stored in the memory are typically larger than 10 Mbytes, and the satellite can on average only be communicated with clearly twice per day for five – ten minutes. It is clear that the high-speed download path is necessary to receive the images efficiently. When an image is sent directly down to earth (not stored in the memory) the real-time speed needs to be maintained because there is no buffer in the downlink data path. The imager and QPSK modulator was designed to work at a maximum frequency of 40 Mbits/s. This is the speed needed at an 800 km altitude, which yields a pixel resolution of 15 m. The satellite was designed for an 800 Km circular sun-synchronous orbit, but due to launch constraints and the orbital demands of the main spacecraft aboard the Delta II launch vehicle, SUNSAT is in an elliptical orbit with apogee and perigee at 800 and 500 Km respectively. With the lower altitude a higher resolution image can be obtained, but to keep the pixels square the bit rate must exceed 40 Mbits/s. The RAMDISK was tested to work up to 65 Mbits/s. The imager was also tested at this speed, but image quality was found to deteriorate. The reason for this is that the CCD voltages does not have time to reach their final level. See 2.2.4 for an explanation of the CCD voltage response. This has the effect that the colour depth of the image is not optimal. At these lower altitudes real time images cannot be taken because of the QPSK 40 Mbits/s speed constraint, however it can still be stored in memory and downloaded at a lower bit rate.

As already mentioned, the S-band serialiser has two data sources. When taking real time images (bypassing the memory) the data is routed directly from the ADC's. A set of multiplexers chooses between the real time path, and data read from memory. See 2.2.5. After the multiplexers have selected the correct source each byte is latched, and then serialised using another multiplexer. A 74AC151 is used, which has eight inputs and one output. A counter is then used to generate three signals, which chooses between the eight inputs. This serial data is sent to a buffer which transmits it to the QPSK modulator using coaxial cable. A synchronised data clock is also sent to the modulator.

2.2.11 Inline FIFO

The option to take single band images exists. This obviously takes up three times less space if the image is stored in memory, and the bit rate required is also three times slower. A single band image is very useful if an area needs to be surveyed to identify

where three band images are needed. For these surveying images quality is not the import factor, but size is. If a large area is covered the surveying process becomes more efficient.

The image size can be reduced and reasonable image clarity can be maintained by using a technique that discards every second line. This is done by using a FIFO that stores the data for two lines, and then only one line's data is written to memory, and the other is discarded.

Discarding of lines only work for single band images. This is not a problem, because if a three-band image is taken all the data is required.

2.2.12 Failed Daughter board management PAL

The RAMDISK's memory is arranged in 8 Mbyte blocks. These blocks take the form of daughter boards, which are inserted into header connectors on the RAMDISK motherboard. As mentioned in 2.2.2, the address switch from one daughter board to the next occurs automatically. A counter, which enables the buffers to each board, is incremented.

In the event of a daughter board failure this automatic switching process will not bypass the affected board, which will effectively divide the memory space of the RAMDISK. The reason for this is that when storing an image only the start address is specified and then the image is stored sequentially from that point. This process can only continue up to the failed daughter board and not beyond.

An intelligent switching process is required to bypass this problem. An algorithm that skips failed daughter boards is implemented in a PAL device. The PAL senses which daughter boards are switched off and then bypasses these when selecting the next daughter board's buffers. The PAL can be switched on or off, and is not used for normal operation.

2.2.13 Compression FPGA

In the main data path a Xilinx FPGA can be inserted by switching on relevant buffers. This is a general purpose FPGA and its mission was not defined before flight. This is

an SRAM based FPGA, which means the configuration must be loaded each time the device is power up. The required configuration can be loaded by software run on the OBC aboard SUNSAT. The function can thus be changed in flight.

The most probable use for this device is to implement a compression algorithm, which will compress the raw data from the imager.

Experiments using this FPGA must still be defined.

2.2.14 Power Control

The power bus of the satellite is 14 V. The RAMDISK uses only 5 V and this is supplied by two voltage regulators. Only one regulator is necessary for operation; the second regulator is for redundancy. The only other place where 14 V is used is with implementing the power switches. These switches use FET devices and the 14 V is used to switch them on. The high voltage on the gate of the FET ensures a small voltage drop between the source and the drain.

Different power switches were incorporated into the design to minimise power wastage. If a subsystem is not being used it is switched off.

There are four main power busses on the RAMDISK:

- Main power
- Motherboard power – This powers all the support motherboard chips and memory. For normal operation when only the memory is required main power and motherboard power is required. Each daughter board can also be switched on and off individually
- Multiplexer power – This switch powers all the multiplexers that routes data from the imager to ram, ram to s-band and l-band to ram
- Imager power – The image power is used for all the imager support hardware such as analog to digital converters. It also provides power for the imager itself.

The main power comes directly from the regulators and powers the circuits that control all the power switches. The telemetry logic is also powered by this source, because the commands that switch the other power switches on is sent by telecommands.

2.3 Faults Encountered and Solutions Found

In this section all the faults encountered will be discussed. Each solution will then be discussed in detail. Section 2.4 covers the software used to debug and test the RAMDISK.

As mentioned in the software section, most of the debugging was done with PC interface cards used to simulate the support structures around the RAMDISK. Two 8255 based cards were used. The one was used to simulate an OBC, and the other one for the telecommand system. These two cards were sufficient for most of the testing and debugging of the RAMDISK. However for complete testing of the image path, the imager was also needed as part of the test set-up. No device to substitute the imager was available. It was therefore necessary to use the actual imager. This did not create problems, as the imager was already operational, making it available for the testing of other systems.

Two identical RAMDISK trays were manufactured; an engineering model, and a flight model. The flight model is the tray that is currently flying aboard SUNSAT. The engineering model is a copy of the flight model, and has two main purposes: to serve as a test bench for possible modifications before they are made to the flight model, and to form part of a ground based test bed for software before it is uploaded to the satellite.

The full set of schematic sheets for the RAMDISK can be found in the SUNSAT RAMDISK documentation. The RAMDISK version that corresponds to these schematic sheets was the final version before flight. Due to time constraints a new RAMDISK would not be manufactured. All the modifications were made by cutting the appropriate pins and adding connections using thin wire wrap. These wires were carefully routed on the board, and secured with a potting agent. All the chips that had any of their pins cut in the modification process were also secured by using this potting agent. On the schematic sheets all the modifications are hand drawn.

2.3.1 Debug and Testing Methods

Due to the sensitive nature of some of the hardware on the RAMDISK careful handling procedures had to be followed to guard against electro-static discharge. These included wearing a grounding wire whilst handling the RAMDISK.

The method followed for testing the different functions on the RAMDISK:

1. Analysis of the RAMDISK schematics.
2. Software development to support or drive the required function.
3. Measurement
4. Comparison of the results and expected response.

Figure 5 contains a flow diagram of the debugging method followed.

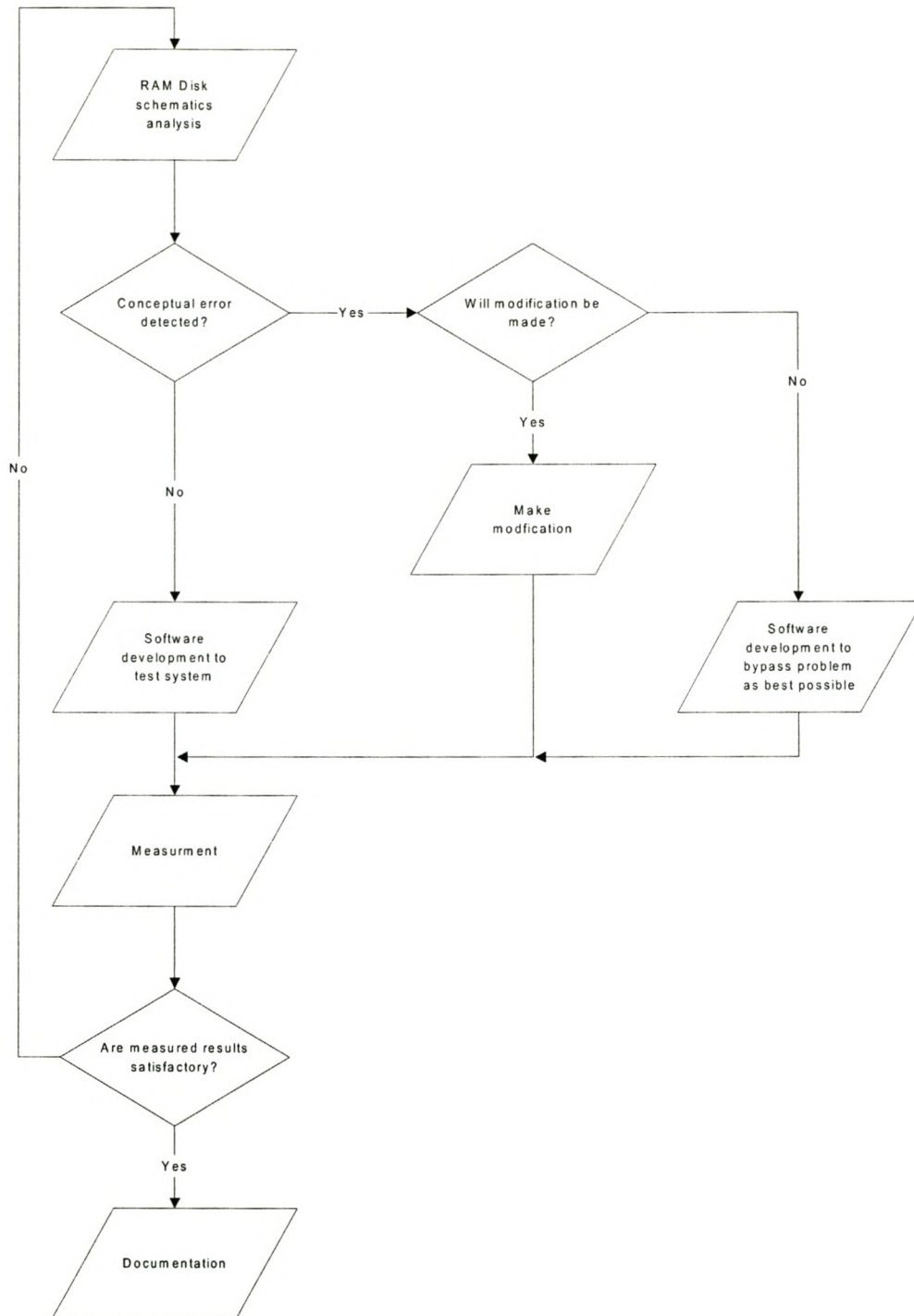


Figure 5 – SUNSAT RAMDISK Debugging Method

2.3.1.1 Analysis of Schematics

The set of schematic sheets for the RAMDISK was thoroughly studied before any steps were taken. An appropriate sketch of the wanted input signals was drawn. Using these as a starting block the corresponding output signals were found by analysing the

circuit. They were then drawn underneath the input signals. Using this graphical method a clear understanding of the circuit was developed.

At this stage one of two routes emerged. The working of the circuit was deemed theoretically correct, and the next step – 2.3.1.2 – was followed. The other situation arose from conceptual design errors. Modifications to the hardware were needed to correct these errors.

2.3.1.1.1 *Modification Considerations*

These modifications had to be carefully considered before they were made. The reason for this is because of the nature of a satellite it has to be exceptionally reliable. Any modifications implies cutting, extra wires and soldering on the RAMDISK. The concerns are:

- From a mechanical point of view this weakens the structure, which undergo huge stresses during launch.
- The extra soldering could cause undetectable errors such as dry joints and degradation of components due to heat.
- This necessitated extra handling of the flight model RAMDISK, which increased the chance of damage by electrostatic discharge.
- During the process of making modifications damage to surrounding structures such as tracks and chips could result.

The possible modifications were first evaluated using the following criteria:

- **Is the design error mission critical?** If this is the case it must obviously be repaired regardless of the risk involved.
- **How big is the impact of the modification?** If the impact on the board is too great another solution must be found. This solution could be a re-layout of the system, which would imply a large time delay, or another path must be identified which would bypass the problem.
- **What is the impact of the design error?** If the mission of the RAMDISK can still be executed with little or no performance loss the modification may be ignored.

2.3.1.1.2 *Modification Process*

If the above mentioned criteria were investigated, and a modification was still deemed necessary the following steps were taken:

1. Design of the modification
2. Qualification of the design
3. Discussion of the design
4. Implementation on the engineering model RAMDISK
5. Final approval
6. Implementation on the flight model RAMDISK
7. Qualification testing

2.3.1.1.2.1 Design of the modification

Here the modification to correct the original design error found during analysis of the schematics, or measurement analysis, is designed.

Due to the nature and diversity of the problems encountered a structured design method was difficult to follow. The original design's circuits and components were already in place, and due to space, weight and structural constraints components could not be added. Thus a re-design was not possible and the current one had to be modified as best possible to deliver the required results. In extreme cases components were added, but this had to be kept to an absolute minimum. Some packages on the RAMDISK had unused gates and these were also utilised.

It is clear from the previous paragraph that the optimal solution could not always be implemented and "jury rig", or patch designs were implemented. Regardless of this the result had to be robust and reliable, which proved an interesting challenge.

Each fault and subsequent fix will be explained in following sections.

2.3.1.1.2.2 Qualification of the design

One possible fix has been identified. It was first tested on a breadboard, or if necessary, a soldered prototype. Here the inputs and outputs could easily be manipulated and measured, and all possible inputs were evaluated.

With this method the behaviour of the circuit could be observed, which will simplify the testing once it is implemented in the more rigid RAMDISK environment. In this environment only defined inputs are available, but with the prototype testing the circuit was qualified for all circumstances.

2.3.1.1.2.3 Discussion of the design

At this stage the fault and proposed modification was discussed with the relevant personnel. These are people who are associated with the SUNSAT project, and have expertise in the field that the modification is made in.

Here the results of the quality testing are shown, and the idea behind the modification is explained. Any conceptual, and design mistakes that were missed are pointed out, and can be corrected before it is tested on the engineering model RAMDISK.

This step is especially valuable as it eliminates the possibility of more errors even further. Experience can be tapped, and lessons can also be learned.

2.3.1.1.2.4 Implementation on the engineering model RAMDISK

Even after thorough testing the modification must still be proven to work in its target environment. Because of the need to maintain the integrity of the flight model RAMDISK, one only wants to make the modification once, and all possible errors must be eliminated. This can be achieved by first testing the design on the engineering model RAMDISK.

Note that although this is the version that will stay on the ground, it must still be used in a support capacity after the launch of SUNSAT. This tray is for testing software, and also crisis management (if a system fails on the satellite, the ground set-up can be used to devise a solution, or at least minimise the impact of the failure). For this reason it must also be handled carefully. In the event that the flight model becomes inoperable before launch the engineering model will be used as a backup. This is another reason to handle it with care.

The modification is made to the engineering model RAMDISK at this stage. The modification is then tested as rigorously as possible.

2.3.1.1.2.5 Final approval

At this stage the original error is presented to the project leader with the proposed modification and test results. The modification implications to the flight model RAMDISK, as well as project schedule is discussed at this point. When approval is obtained the modification is made to the flight model.

2.3.1.1.2.6 Implementation on the flight model RAMDISK

The modification to this tray may only be done by qualified personnel. This is to ensure that the soldering is of sufficient standard, and also that minimum damage to the flight model is done.

The RAMDISK circuit board has four layers. Removing components is hazardous because the inner layers, or via connecting the different layers could be damaged. The process of modification has thus got to proceed with the utmost caution. The other problem is that much of the damage that could result is very difficult to detect, and only starts degrading system performance after a period of time, or after long exposure to the harsh space environment. This is why qualified personnel are necessary.

2.3.1.1.2.7 Qualification testing

Here the flight model RAMDISK needs to be inspected and tested to see if the modification is of sufficient quality, and whether no other systems were damaged during this process.

All the tests performed during the initial stages are now repeated, as well as tests for any systems that were impacted by this modification. The physical surrounding circuitry is also tested to detect any damage that might have occurred during the modification process.

When this testing is complete the schematic sheets are updated to reflect the change, and any documentation necessary is done.

2.3.1.2 Software Development

After the schematic analysis, software must be developed to test the working of the systems and to verify correct operation. In case a modification is needed the software is developed during the process described in 2.3.1.1.1 and 2.3.1.1.2.

As mentioned in 2.4 many of the software was already in place. But due to the demands of modifications, and also more thorough testing, the software had to be modified and some new code added.

This development stemmed directly from the analysis of the schematics, and test or operation procedures which are developed to qualify the RAMDISK.

2.3.1.3 Measurement and Testing

Once the software has been created the circuit under scrutiny is tested.

The maximum number of scenarios possible is simulated. This is to ensure that all the possibilities are covered.

2.3.1.4 Comparison of results

The behaviour measured during the testing phase is compared to the behaviour expected from the analysis of the system schematics. If any discrepancies occur the schematics are re-analysed to see if a conceptual error was made.

If this is not the case the hardware must be carefully debugged until the discrepancy is explained. Here two possibilities emerge. One is that a hardware failure has occurred, and the other is that a conceptual design error is discovered.

If a conceptual design error is detected the process explained in 2.3.1.1.1 and 2.3.1.1.2 is followed. If a hardware failure is detected the corresponding circuit on the engineering model is tested, and the components that need to be replaced are identified. If the error only occurs on the flight model the faulty component is replaced, and if it is found on both trays the reason for this failure is explored and a solution found.

2.3.2 Final Qualification

This consists of writing software and procedural documents to describe a number of tests to perform periodically to verify the integrity of the RAMDISK.

These procedures are followed, and any error results normally have a predefined solution.

2.3.3 SUNSAT RAMDISK Design Errors

In this section all the errors and conceptual design mistakes will be discussed. Please refer to the SUNSAT RAMDISK documentation for the full set of schematic sheets. The relevant circuit diagrams will be included in the text.

2.3.3.1 L-Band Uplink Serial to Parallel Converter

The problem with this circuit sits with the LRX_PARCLK, or L – Band parallel clock. This clock is used as the data clock for the byte received via L – Band. The data is serialised with a shift register and the incoming data clock. This incoming data clock also drives a counter that creates a pulse every eight clock edges. This pulse serves as the LRX_PARCLK, and also to reload the start value of the counter with 7, ready for the next count. But as soon as the pulse is created the counter is loaded, which clears the output. Thus the pulse is not a defined length, but a glitch.

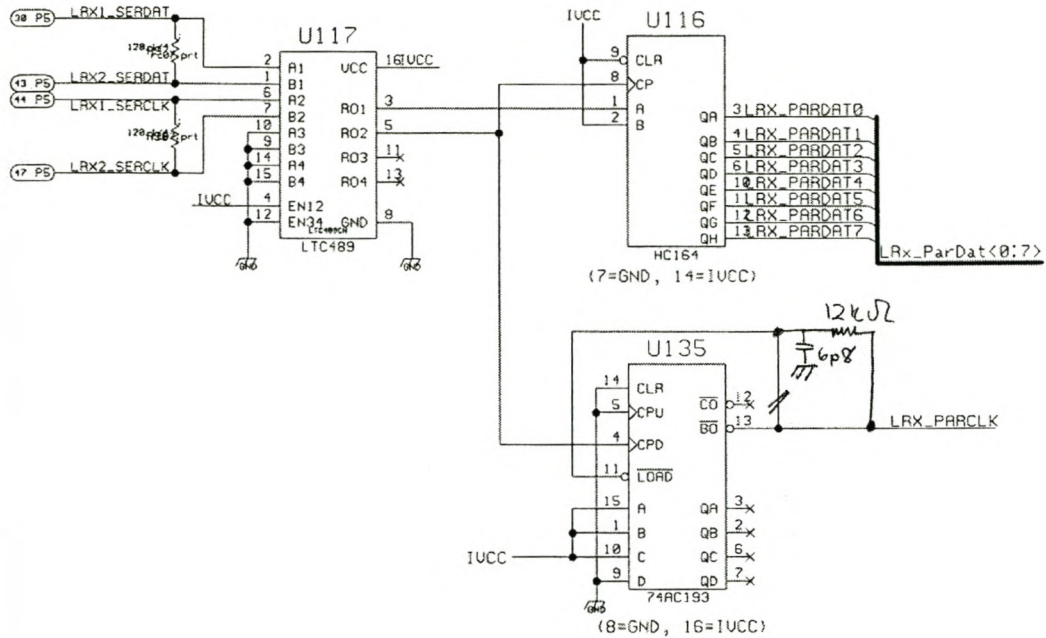


Figure 6 – L-Band Serial to Parallel Converter

2.3.3.2 Clock Generation Circuit

The crystal oscillator packages are very stable, but the inverter oscillation circuit is not. And due to radiation exposure the oscillator will also change in frequency.

The clock extracted from the L- Band receiver was used instead.

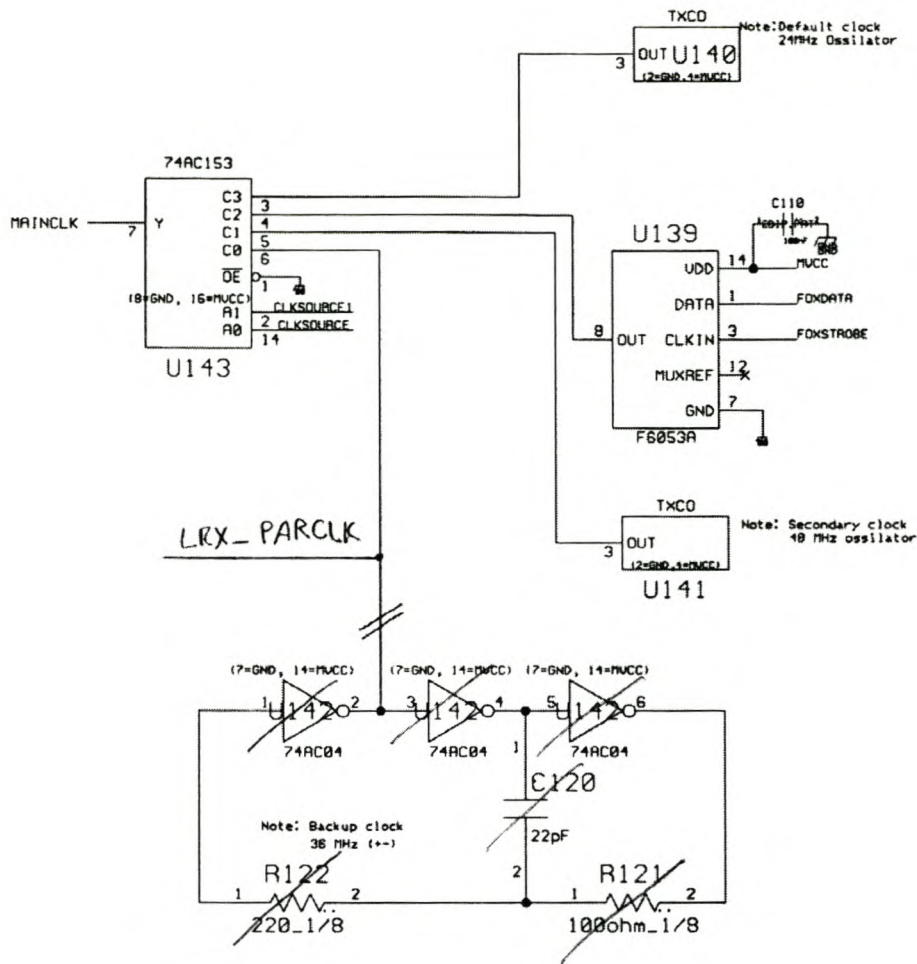


Figure 7 – Clock Generation Circuitry

2.3.3.3 CCD Clocks

The system clock is divided and manipulated in this circuit to create the correct pulses for the CCD sensors.

The problem occurs with the generation of the sample pulse that is used by the ADC to sample the CCD voltages. The CCD voltages have a voltage curve resembling a discharging capacitor. The final voltage level to which it discharges depends on the amount of light that is sensed by the CCD element. The optimal point for sampling the voltage is just before the next elements voltage response is received. Due to the different altitudes of SUNSAT the rates at which images are taken are not constant. This meant that placing of the sampling point is very critical, because a point that gives

the correct value at lower speed, when the CCD has enough time to reach its final value, may not do so at higher speeds.

This is what was wrong with the original sampling point. Initial tests at 25 MHz (rate at which pixels are received) gave a clear image. The operational data rates of 33-60 MHz were never tested in the design phase. When an image was taken at higher data rates during qualification testing by the author it was determined that the sampling point was too close to the reset pulse of each CCD element, which caused degradation in images taken at higher speeds.

The correct position was attained by delaying the sampling point as much as possible; just before the next CCD elements' information arrived.

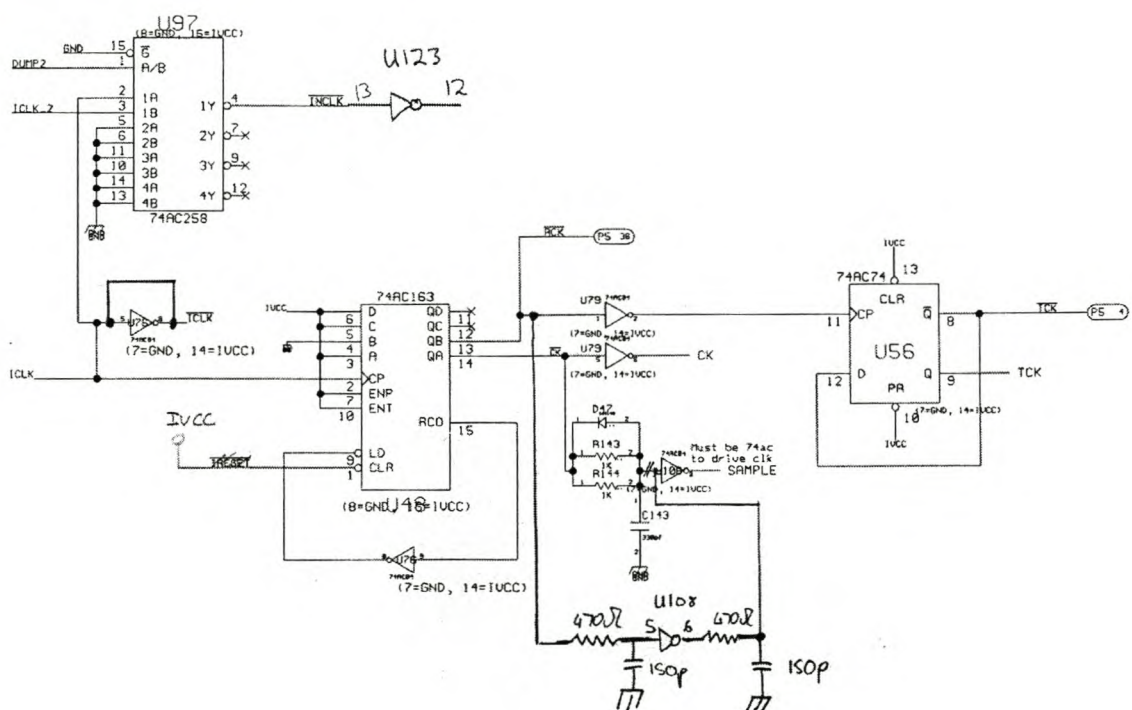


Figure 8 – CCD Clock Generation Circuit

2.3.3.4 Telecommand Interface

The all the telecommand lines have serial resistors on the RAMDISK. These inline resistors protect both the Telecommand system and RAMDISK from short damage. If

a short occurs on one of these boards the resistors will limit current flow, preventing damage to any connected logic.

Some of the lines on the RAMDISK demanded more current than what the current limiters were designed for. The specification was one CMOS input, which draws very little current. The lines to program the FOX variable oscillator, as well as other system, draws much more current than one CMOS input. The 100 k Ω current limiting resistors on these lines caused a large voltage drop, and the signal was not recognised as a high by the target logic. They were replaced with 3k3 Ω resistors, which still affords short protection, but does not drop too much voltage. Figure 9 show which resistors were changed.

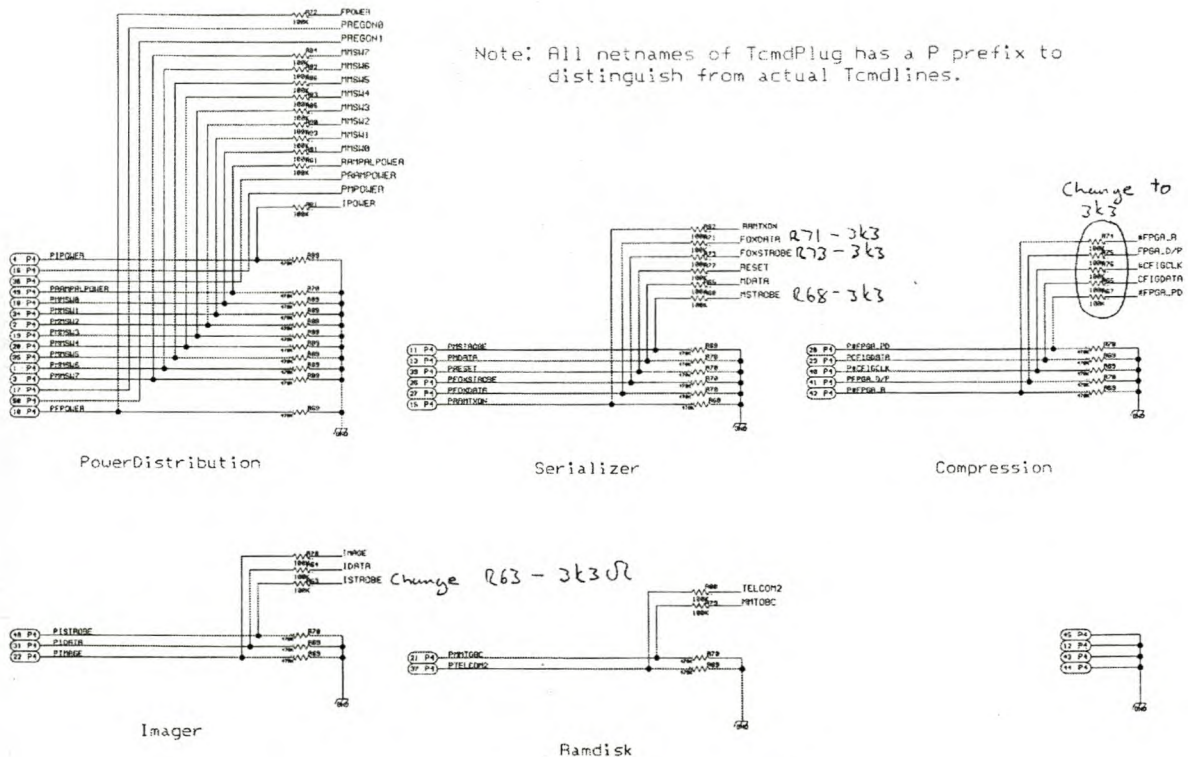


Figure 9 – Telecommand Interface

2.3.3.5 Reset Circuitry

The reset line for the BUSY signal was found to be incorrect. The BUSY signal is used to indicate when image data must be written to the memory. When an image is to be stored in memory, or sent to the S-band serialiser the IMAGE signal is taken high,

which connects the XCK signal to the flip-flop that sets and clears BUSY. The XCK signal's rising edge coincides with the start of a new line of image data. This rising edge sets the BUSY signal.

The BUSY line is reset once the line counters has expired, or a reset is received. This reset signal was found to be incorrect, as it blocked the pulse which resets the BUSY signal. /MRESET is normally high, and low during a reset. In its normal state the lines expired pulse is not blocked, and in the reset state the BUSY signal is cleared. Figure 10 indicates where /IRESET was replaced with /MRESET.

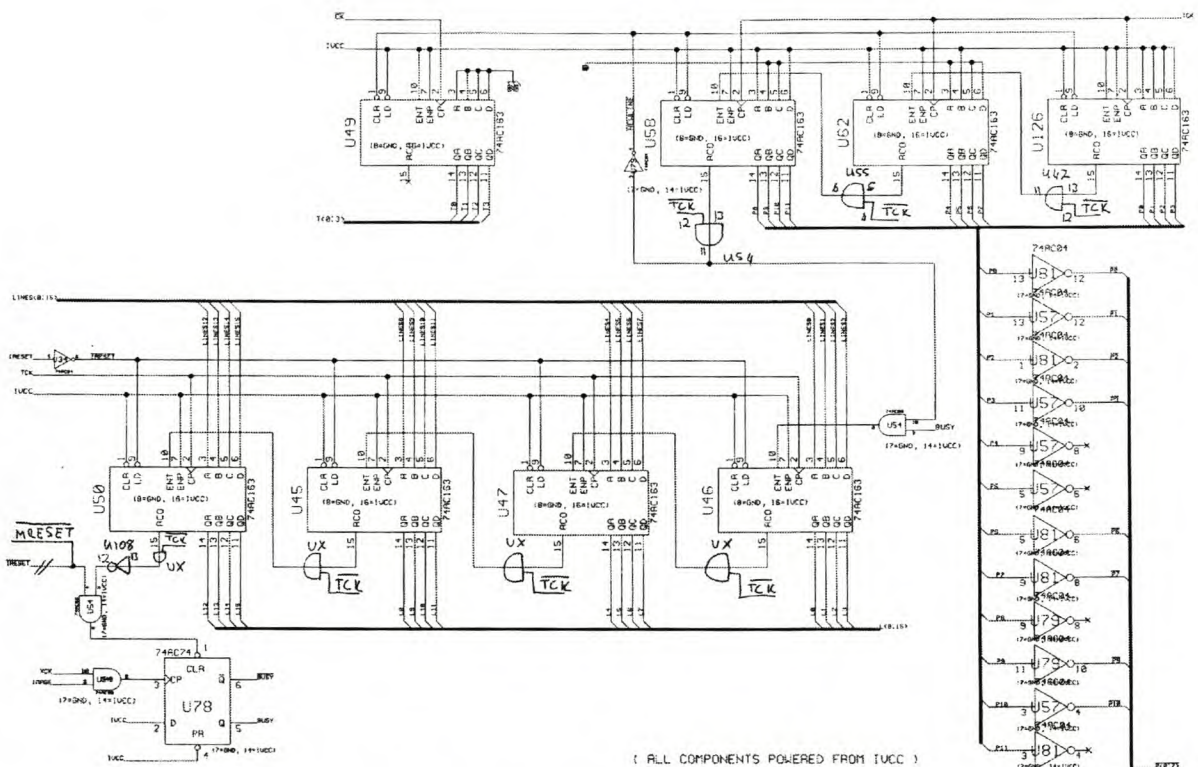


Figure 10 – Line and Pixel Counters

2.3.3.6 Line and Pixel Counters

This modification was made by Peet Badenhorst, and is covered in his thesis document [3]. It was found that the line counters did not always count the correct number of lines. The line length count also displayed glitches.

Figure 10 indicates his solution. The overflow lines were AND'ed with /TCK. This blocked the glitches, and the counting proceeded normally.

2.3.3.7 FIFO

The FIFO was inserted into the data path after the design was completed. Its function is to remove every second image line. This is useful if a large area needs to be surveyed, and high image quality is not a requirement.

Telecommand signals were added to operate this FIFO. This component of the RAMDISK was never tested, and the conceptual errors in operating the FIFO was not seen. Only when final testing by the author was done were these problems highlighted.

There were two major mistakes; the FIFO was not given the correct control signals, and it was too small.

The correct control signals were routed to the correct pins using an AND gate, and the BUSY was employed to ensure that data was only written to the FIFO at the start of a line.

The size problem caused 20% of the image to be lost. This was not seen as a big problem, as the remaining 80% was correct, and the nature of the FIFO caused information to be discarded. Thus the risk of removing the chip from the motherboard was not taken.

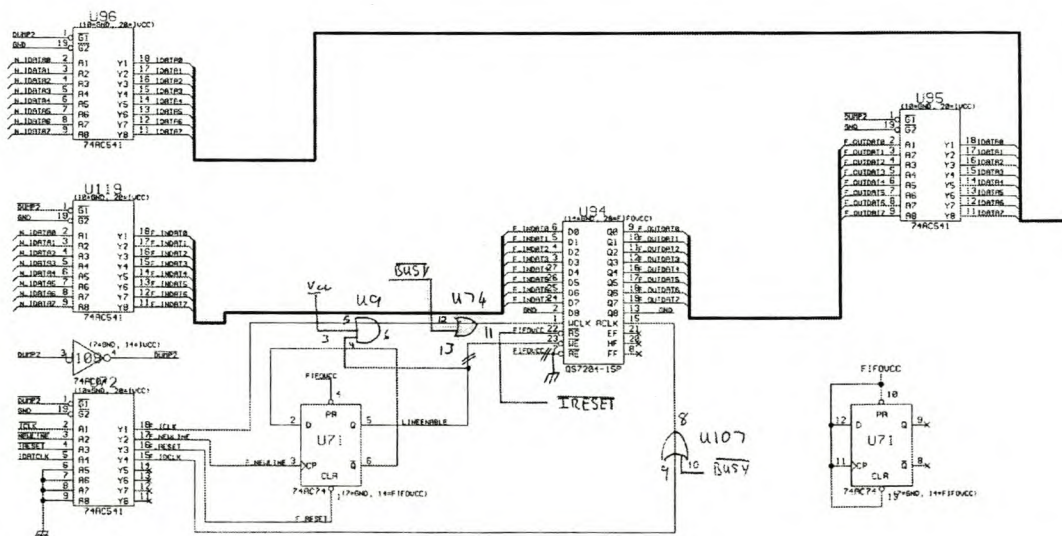


Figure 11 – FIFO

2.3.3.8 Source Switching Problem

There are four possible sources where data can flow from/to. The two OBC's, the imager and the S-Band downlink. A decoder with two inputs, and four outputs selects the appropriate source. Each output enables a specific source, and is selected by the correct combination on the input lines. These input lines are telecommand lines.

It was found that if the correct sequence was not followed, address glitches occurred. The fix for this problem was done in software. By switching the correct telecommand line first, the erroneous state was not entered, and the address was not disturbed.

2.3.3.9 PAL Program

The PAL handles the switching of daughter boards in the event of a daughter board failure to keep a continuous memory space. The PAL program was created with CUPL, a software package that creates the required net-lists for PAL and GAL devices.

The PAL that was soldered into the board did not perform the required function. The program with which it was loaded was also not available. Seeing as it has quite an important function the device was removed from the board, and a socket put in its place. A previous version was loaded and tested. When final qualification was complete the device was potted into the socket to ensure structural integrity.

2.3.3.10 Black Reference

The CCD sensor in the imager supplies a white and a black reference. These references are to calibrate the normal image information. This white reference occurs at the start of a line, and the black references are situated from pixel number 10 to 16. A signal BCLAMP must be sent to the imager to indicate that the black reference must be sampled. This signal started at after the wrong amount of pixels, and did not yield a satisfactory black reference.

This problem was fixed by changing the decoder combination that sends the BCLAMP signal to the imager.

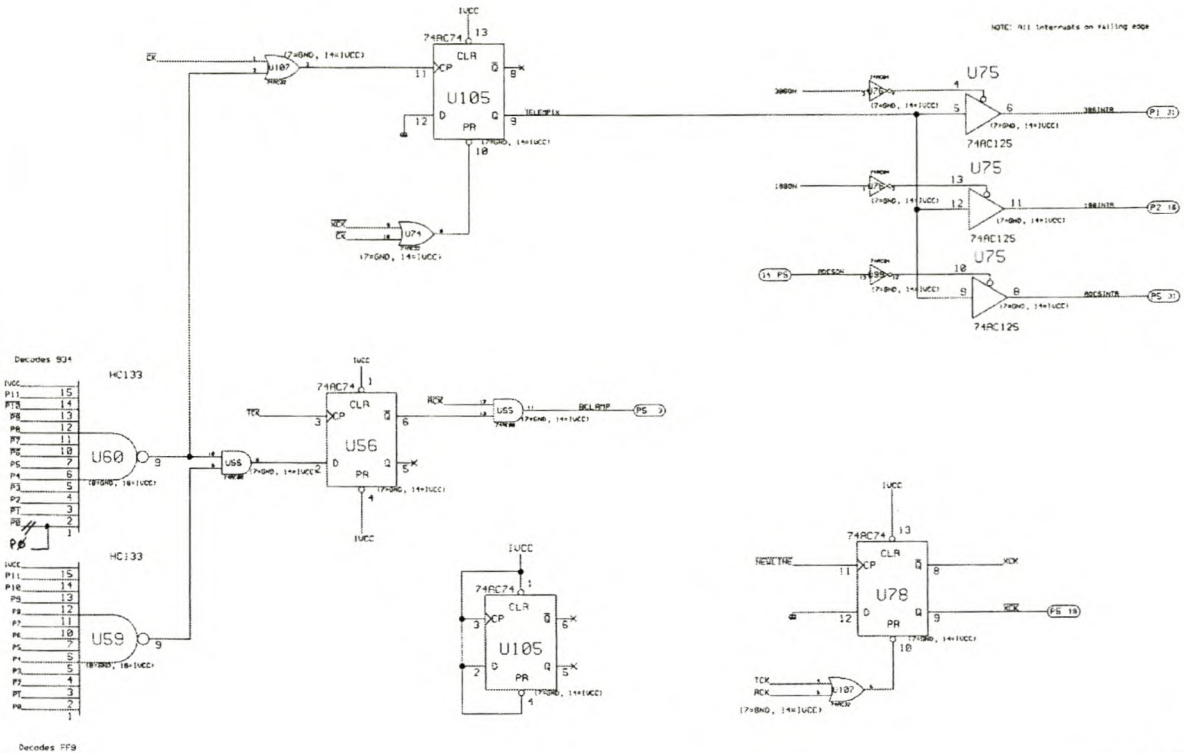


Figure 12 – Interrupt and Black Reference Generators

2.3.3.11 S-Band Serialiser

The S-Band serialiser presented one of the most challenging problems on the RAMDISK. Sending of image data directly down to earth, or real-time (not storing it in memory first) was also a requirement that was added after the design was completed. The result was that this function did not work at all.

Data that was read from the memory was properly synchronised, but the serialisation process did not correctly send the bytes of image data down to earth. The most significant and least significant nibbles were swapped, which had the effect that bit 4 to 7 were sent, and then bits 0 to 3.

Data that was sent real-time was not synchronised at all. Nibbles from different bytes were mixed.

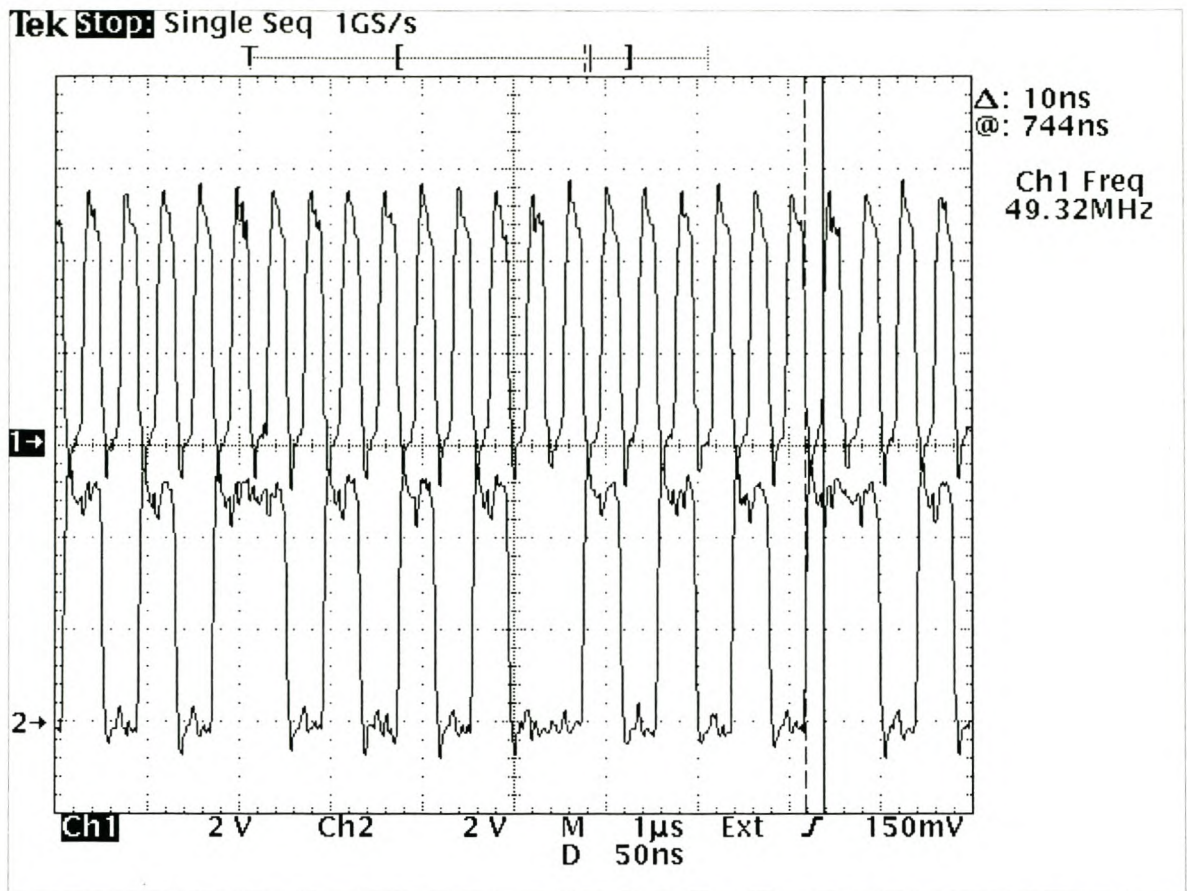


Figure 14 – S-Band Serialiser Measurements

2.3.3.12 Analog to Digital Converters

Figure 15 shows the order in which the three colour bands are stored in memory. This backward order results from the shift in sample point.

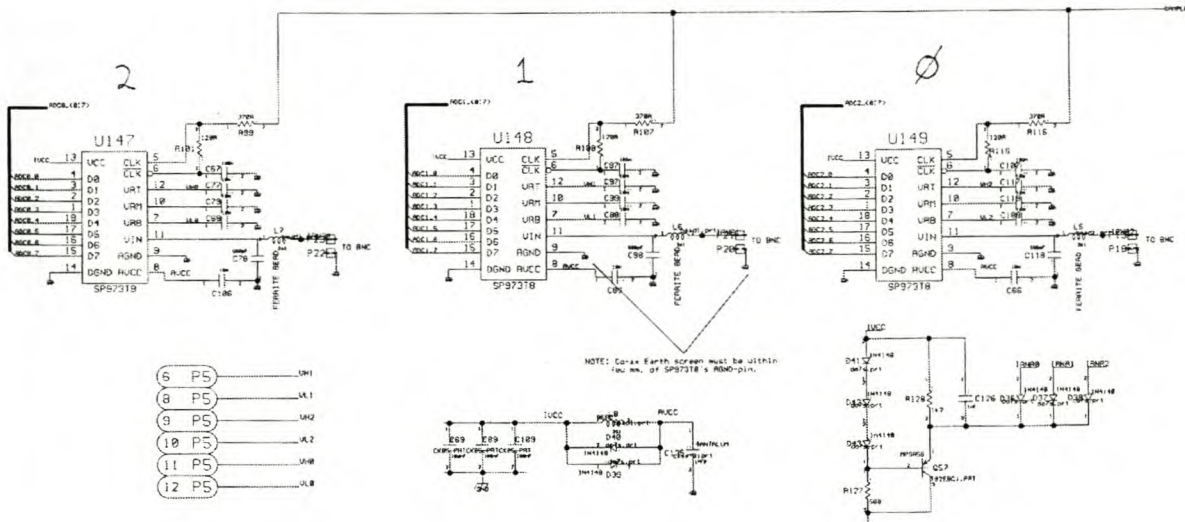


Figure 15 – Analog to Digital Converters

2.4 RAMDISK Software

The RAMDISK is highly software dependant. Most of the intelligence is implemented by software, which will run on one of the OBC's on SUNSAT. The commands on the RAMDISK are set via telecommand, and data transfers and address set-up procedures via the OBC – RAMDISK interface. One port on the RAMDISK is dedicated to telecommand lines, and two ports are used for the OBC interface, one for each OBC. The telecommand interface is covered in section 2.2.7.

Testing and debugging of the SUNSAT RAMDISK was done mainly by running software commands, and then observing the appropriate circuit. Besides the fact that the OBC boards were not readily available as platforms to run the testing software on, the time required to upload code and boot these systems were prohibitive for effective debugging. The telecommand board would also have to be used. An alternative system was found to simulate the telecommand and OBC systems. This took the form of two PC interface cards, which slots into a PC's ISA bus. 8255 I/O IC's acts as the ports, and the appropriate signals are set on these IC's ports to simulate an OBC, or telecommand system.

The same high-level software is used on both the PC test-bed and the real OBC. The only difference is in the low level driver for each system. A file containing flags determine which section of software is compiled. This file also contains flags for other

compile conditions such as whether the flight model RAMDISK or engineering model RAMDISK is used. The only difference between these two boards is the daughter board size. On the flight model it is 8 Mbytes, and 2 Mbytes on the engineering model.

The Modula 2 programming language is used for all the high-level software regarding the RAMDISK, and some of the low level drivers are written in assembly language.

The testing software for the RAMDISK was derived from code written by students who worked on the RAMDISK previously. Many structural changes were made, and conceptual errors fixed. Previous code was written by P. Badenhorst and H. Burger, both of whom worked on the RAMDISK.

2.4.1 Software Functions

The software here was written mainly for testing the RAMDISK's functions. Procedures were also written for automated periodic testing to ensure that the system is still performing normally.

These software procedures were ultimately also built into the flight software, but with a different structure.

The definition file contains the interface to all the procedures. This will be explained here and in doing so the software functions will also become apparent.

The following descriptions are for the enumerated types created to send the appropriate commands to the RAMDISK.

Ram Power Modes:

- MotherBoardOn - Option of switching on regulators 0 or 1
- MotherBoardOff - Option of switching off regulators, and daughterboard power
- MemBoardsOn, MemBoardsOff - Switches indicated which Daughter boards to switch on and which must be off

- RamPowerCntrlT – This parameter has four possible enumerated types: NoAction (Do not change power status), OnlyReg0 (Only change regulator 0 status), OnlyReg1 (Only change regulator 1 status), OnlyVcc (Only change RamVcc(Daughter board power)), Reg0andVcc (Switches on regulator 0 and mother board power) and Reg1andVcc (switches on regulator 1 motherboard power)

Ram Driver Modes:

- CameraToSband - Indicates that a real time image will be taken => Image data to is sent directly to the S-Band transmitter.
- CameraToRam - Indicates that the image will be stored in memory.
- RamToSband - Indicates that the contents of the memory will be dumped to ground via the S-Band transmitter.
- LbandToRam - Indicates that data will be uploaded to the ram via the L-Band receiver.

Clock Sources

- LbandClk - Clock extracted from L-Band data path.
- TwtyFiveMhzClk - Oscillator in U141 (25 MHz)
- FoxClk - FOX F6053A in U139, this clock is programmable from 360 kHz to 120 MHz. To program the FOX F6053A these four parameters need to be calculated on the ground. Use the FOX\FOXPARAM.EXE program to get these.
 - FoxParameters = RECORD
 - P_ = SHORTCARD
 - Q_ = SHORTCARD
 - M = SHORTCARD
 - Index = SHORTCARD
- FortyMhzClk - Oscillator in U140 (40 Mhz).

Parameters passed in RamDrvCmd

- NumLines - Number of lines for image.
- NumColours - Number of colours for image. (one or three)
- Channel - Used if a single colour image is taken. Selects colour for image (channel 0/1/2)
- SbandDivisor Default value = 7 This selects the divisor of the clock used by the S-band. Values are: 0 = DIV 16, 1 = DIV 8, 2 = DIV 4, 3 = DIV 2, 4 = DIV 32, 5 = clock DIV 4, 6 = clock DIV 2, 7 = clean clock. The default is clean clock = 7. Another divisor should be chosen with care, due to modifications in the S-band path the clock divisor system is obsolete.
- UseFIFO - Use FIFO to discard every second line of the image. This implies that the data rate is halved and a more reliable S-band downlink can be achieved. Also used to obtain square pixels in the image. The last 15% of every line is corrupted due to the FIFO being too small.
- UseInlineTelem - Inline telemetry is 12 bytes of data that is written into every line of the image.
 - bytes 1-2 : Number of lines in image.
 - bytes 3-4 : Number of current line (counts down to zero).
 - bytes 5-8 : Data written by ADCS.
 - bytes 9-12: Data written by OBC.
- InlineObcData - Data that is to be written into the picture by the OBC's Eg : data := (byte1, byte2, byte3, byte4)
- StartAdr - Starting address when writing to or reading from ram.
- ClockSource - Selects the main clock on the ramtray. See type definition.
- UsePAL - The PAL is used to create a continuous memory space if one or more of the daughter boards have failed. This is essential when taking an image. The PAL skips boards which are not switched on, thus if a board is faulty the power to it must be switched off and it will be skipped. If the start address is not on the first board the PAL needs to be manually incremented till the correct block is selected. This is done by the address set-up procedure, but the amount of OFF boards must be stated in DeadBoards.
- DeadBoards - This variable indicates the number of OFF banks between bank 0 and the desired start bank. NOTE: Bank 0 is excluded from this number, thus if bank 0 is off it is not added to DeadBanks.

- UseFPGA - Use the FPGA to perform realtime processing on the image.

2.5 Conclusions and Lessons Learned

The tray known as the RAMDISK is actually a collection of different systems included on one circuit board. These systems are highly integrated and no clear separation is defined anywhere. The imager clocks and pulse generators are also spatially separated from the imager, which cause signal degradation, and also make the system very complex.

Extra functionality was added after the design was completed, which also caused problems. These "add-ons" did not always fit into the complex RAMDISK design. Incorrect timing concerning the data paths was a major problem.

Power management is also an important issue. In order to download data from the RAMDISK to ground via the S-Band downlink, the image power had to be switched on. This was needed to drive some of the logic, which routed the data from memory to the S-Band serial link. This is a waste of precious energy, because the imager draws 2 amps, and it is not being used to take images during the download operation.

Another power related problem is the driving of dead logic. This happens when any of the daughter boards are switched off. Their buffers are also switched off, but it's input lines are being driven by logic signals. Fluctuating power usage is caused by this.

The RAMDISK on SUNSAT is also highly software dependant. The driver that is used to operate the system controls all the functionality, and very little automation is included. This makes the software very complex, and also requires that the person writing the software for the satellite have intimate hardware knowledge of the RAMDISK.

Chapter 3

MEMORY DRIVE

3.1 Introduction

The aim of this thesis is to create a storage device that is not only for use aboard satellites but also ground-based applications. An attempt will be made to make the design portable, thus the only major change between an earth and space based realisation of the storage device will be component quality and type. For example an earth-based application can use SRAM technology (volatile, thus upgrades are easy), while in space anti-fuse technology will be more radiation tolerant.

The focus of the new storage device is modular. A routing system, or memory drive, will be designed which will include all the features necessary to interface between a high-speed serial bus and memory modules. The memory modules will contain the actual memory devices and depending on the technology used a processor that will service the memory. This processor can be implemented in a programmable device such as an FPGA and will maintain the standard interface to the memory drive. In the case of DRAM this processor will not only write the data to the memory, but also take care of the refreshing, or with Flash it will manage the complex writing algorithm. Here the interface between the central routing system and memory modules will be designed. These will be called memory interfaces. With each implementation a type of memory will be selected, and the appropriate logic to drive it will be added to the memory interface.

Another very important aspect will be to simplify the software needed to control the memory drive. Using VHDL a lot of the complexity normally associated with software can be implemented in hardware. A standardised set of instructions can be defined, and the software engineer only needs to know these to write an effective driver for the memory drive.

3.2 Design Discussion

The main function of the new memory drive will be to store images taken by the onboard imager. These images will normally be raw data directly from the imager, which means that a large amount of data must be captured at high speed.

It will also be used to store general satellite data. This may be as a primary or backup memory device, depending on how much memory the on board computer (OBC) has located locally. Here speed is not so important, because of the limited speed at which the OBC can read and write data.

Design criteria include:

- Speed
- Memory size
- Reliability
- Complexity and redundancy
- Power consumption
- Physical size
- Cost

3.2.1 Speed

Imaging satellites are typically in low earth orbit (LEO) to capture high-resolution images [15]. LEO satellites have high ground speeds (due to its orbit), thus CCD sensors must operate at high frequencies to keep the pixels square. The width of the pixel is determined by the CCD element size, but the length depends on how fast the CCD samples, too slow and the pixels are elongated, too fast, compressed. The data generated by the imager has to be stored in real time and fast enough in order that when the CCD takes another sample the previous data is already stored. Also the imager will have more than one colour band, which will multiply the required data rate by the number of bands.

The captured data will be downloaded to a ground-station via a high-speed link. But this will be slower than the required capture rate, thus the writing speed must be the determining factor.

The memory drive will have to support two major forms of data access, random and sequential read and write operations. When the OBC will use random read and writes. This means that the address needs to be set up before each read, or often in a read/write process. When capturing data from the imager the address will only be set up once, and then the data will be stored sequentially. The speed requirement must be met for capturing data, and the drive will be designed for sequential operation.

3.2.1.1.1 *Speed Requirement*

The next generation imager has not been designed yet, but a proposal has been put forward. In order to obtain a resolution in the 5 meter per pixel range and using current CCD technology 13 Mega-samples per second need to be captured. Each sample will have a 10 bit resolution. This gives a required data rate of 130 Mbits/s. Note that this is the rate for one colour band.

This data rate may vary, as it is highly dependant on the type of CCD used, the image resolution, etc. For this design 130 Mbits/s will be taken as the speed requirement.

3.2.2 Memory Size

As already mentioned the purpose of this memory drive will be to store images. These images will be in raw format (as received from the imager). Due to the speed requirement effective inline compression will be extremely difficult and expensive. The images will thus be stored in raw format, which requires large amounts of memory

The images can only be retrieved if the satellite is in contact with a ground station. LEO satellites are typically only visible two to four times in a 24-hour period [15]. The result is that if the memory drive is filled to capacity imaging can only resume once the data has been downloaded to earth. It would be prudent to have space to store at least four images.

Because an imager has not been designed yet the following parameters will be assumed (these are in-line with the current imager proposal):

CCD sensor:	10000 elements wide
Sample resolution:	10 bits
Image resolution:	5 meters per pixel
Image width:	$10000 \times 5 = 50 \text{ Km}$
Image length:	500 Km

Table 1 – Proposed Imager Parameters

$$\text{One line} = \frac{10000 \times 10}{8} = 12500 \text{ bytes}$$

$$\text{There are } \frac{1000}{5} \times 500 = 100000 \text{ lines}$$

This means that for an average image 1.25 Gbytes of memory will be needed to store one band's data.

The memory drive will be modular, which means that any amount of memory can be added, dependant on the specific mission. For the proposed second generation imager 4 Gbytes per band will be needed.

3.2.3 Reliability

Reliability is an important issue and has two major influencing factors. One is component quality, and the other is design. Component quality will be further explored in 3.2.6, and design in 3.2.8.

The memory drive is a high priority sub-system aboard a satellite. If it fails the imager also becomes compromised. Depending on the configuration of the satellite a direct downlink to earth (image data sent directly to the downlink modules) can still bypass a memory drive failure, but this will limit the satellite to only taking images while in contact with a ground station, which is a very small percentage of the earth's surface.

Another aspect of reliability is that once the data is stored in the memory it has to stay there until it can be downloaded. One of the biggest problems with memory integrity

in space is single event upsets (SEU). This occurs when radiation causes a single bit to flip. With images this is not a big problem, as a pixel error here or there is not very noticeable. These errors can also easily be fixed by using the bordering pixels to extrapolate a value. If SEU's become common though, it will pose a problem.

3.2.4 Power Consumption

Power is a valuable resource on a spacecraft. It is obvious that the memory drive needs to be as energy efficient as possible.

The mission of the drive will require it to be active most of the operational life of the satellite. The reason for this is that images have to be stored until they can be downloaded. As soon as there is space on the drive this will be filled with another image as soon as possible to ensure optimal use of the spacecraft. The drive may also be used to store various forms of satellite data.

In terms of memory choices there are two options: volatile and non-volatile memory. Volatile memory loses its contents when powered down, dynamic RAM and static RAM falls in this category. On the other hand, non-volatile memory only uses power to write or read data, and keeps its data once the power is removed, Flash memory and EPROM's are classified as non-volatile. The focus of the memory drive is on a modular and generic design. This concept focuses on the ability to interchange memory types, and still have the same functionality from the memory drive. For a thorough discussion on memory technology see [3].

From the above discussion it is clear that a low power mode is necessary. This mode can be entered when no memory activity is taking place. In case of volatile memory power must still be supplied to the memory to keep the data. With non-volatile memory this is not necessary. Status information and other data will have to be stored in this low power mode.

Besides this low power, or standby mode, other varying degrees of power usage must also be available. The full functionality of the drive will not always be used, and components not involved in a data transfer can be switched off.

A very important aspect of power control is to avoid driving dead logic. Driving dead logic occurs when an active device has its outputs connected to an inactive, or powered down device. The active device drives the connecting lines high or low. The in-active device acts as a load, and the problem lies therein that the load is not defined. Often what happens is that one of the input lines causes the in-active chip to power-up. A current path is found between the input and the Vcc pins of the chip. This is normally a high resistance path, but unpredictable behaviour will result. The driven devices may also start to float and oscillate causing more unpredictable behaviour further down the path. If a number of powered down devices are driven an excess amount of current will be drawn, which results in wasted energy, and may also cause failures. The power design must carefully be analysed to ensure that this situation does not occur.

3.2.5 Physical Size

On any satellite based design size is also an important consideration. For a memory device the main constraint is the amount of memory used. The weight will also be influenced by this factor.

This thesis covers the conceptual design of a memory drive that will be highly modular. A range of memory sizes can be used, and the physical size will be determined by the amount of memory that is used in a final implemented of the design.

3.2.6 Cost

The SUNSAT program does not enjoy an infinite budget, and cost must also be considered. To minimise the chance of components failures military grade and radiation hardened devices are normally used in spacecraft. The higher reliability of the components comes at a price, and these military grade devices are very expensive.

Due to the very limited budget of SUNSAT most of the components used were commercial grade, or "off the shelf." This has proven to work in space, but the length of reliable operation is still unknown.

The choice of components will be highly dependant on the mission length, type and budget. Once again the statement is made that this thesis is a conceptual design, and

the above-mentioned parameters are not defined. Once the exact mission specifications are known the component choices can be made and the design implemented in these.

3.2.7 Redundancy

Redundancy deals with the issue of failures aboard the memory drive. When a failure occurs the optimal situation is where a redundant path exists that will continue the mission without loss of performance.

It is clear that redundancy is very desirable, as components often fail due to the harsh space environment. The negative side of a redundant path is that it pushes up the cost, weight, size and power consumption.

The effect of cost, weight and size can be minimised by identifying the critical data paths on the memory drive and partially or fully duplicating these. All the non-essential items are not duplicated.

Power consumption of a highly redundant design can be managed with intelligent power control. Referring to section 3.2.4 different states of power usage will be available. This concept can be extrapolated by assigning any redundant paths to different power groups. Thus they are only switched on when needed.

The memory drive must also be designed with redundancy in mind. Data paths and components must be designed to have more than one use. If a component's primary function is not utilised in an action, and the action's main path has failed, the device can be rerouted, or reconfigured to perform the needed function. This interchangeability of devices must not sacrifice performance, as the memory drive will need to meet speed specifications. See 3.2.1 for a discussion concerning performance. The inter-changeability will rarely be possible to implement, but where possible it must be utilised.

3.2.8 Complexity

The complexity theme ties in with all the other factors discussed above. As a rule the complexity of a design must be kept as low as possible. A complex system is more prone to failure. Size also increases, as well as weight, cost and power consumption.

The problem is that as soon as more functionality is added to a design the complexity usually shoots up. Especially when redundancy is taken into account. Many extra components will have to be used just to manage the transfer of control and data from one system to the next. A compromise must be made between complexity and redundancy.

Another aspect, which will increase the complexity dramatically, is automation of the memory drive. With the RAMDISK on SUNSAT most of the control is exercised by software, which is run on one of the OBC's. This software dependency results in very complex software, and ties up OBC resources. Each system on the SUNSAT RAMDISK has a different interface, and is accessed or activated by different means. No structured interface exists. The complexity in the software makes it difficult to maintain and upgrade. Software bugs may easily also go undetected and cause problems.

With the memory drive the idea is to shift much of the intelligence to the hardware, which will simplify the software, and also give it a more structured interface. This interface will be well defined, and a single driver can be used to control the drive.

The shift of intelligence to the memory drive will cause a sharp increase in complexity, which comes with all the problems described above. This is deemed necessary, as the software situation cannot continue as is.

A compromise must be found between functionality and complexity.

3.3 Design Overview

There are three main aspects to the memory drive. Please refer to Figure 16.

- Central Data Router
- Generic Memory Interface
- Command Register

The support components are:

- Memory Interface Power Control
- Parallel To Serial Converter
- Serial To Parallel Converter
- Communications Processor

Each component will be discussed separately, but it must be noted that they are highly integrated, and have specialised interfaces. The thick lines in Figure 16 denote the main bus structure (32 bits), which will be discussed in 4.3.1.1.

There are three serial busses depicted in Figure 16. The CAN bus will be used to send telecommands, and serve as an interface to the OBC. More information on the communications processor and CAN bus can be found in [8]. The two fast serial busses will be used to interface with the imager and also to download data to a ground station via a high-speed link. The type of serial bus that will be used is still under investigation, and it is possible that only one high-speed link will be used to transfer data. This project assumes that the data is already decoded, and available as a serial stream with a clock, which means that any serial scheme may be used.

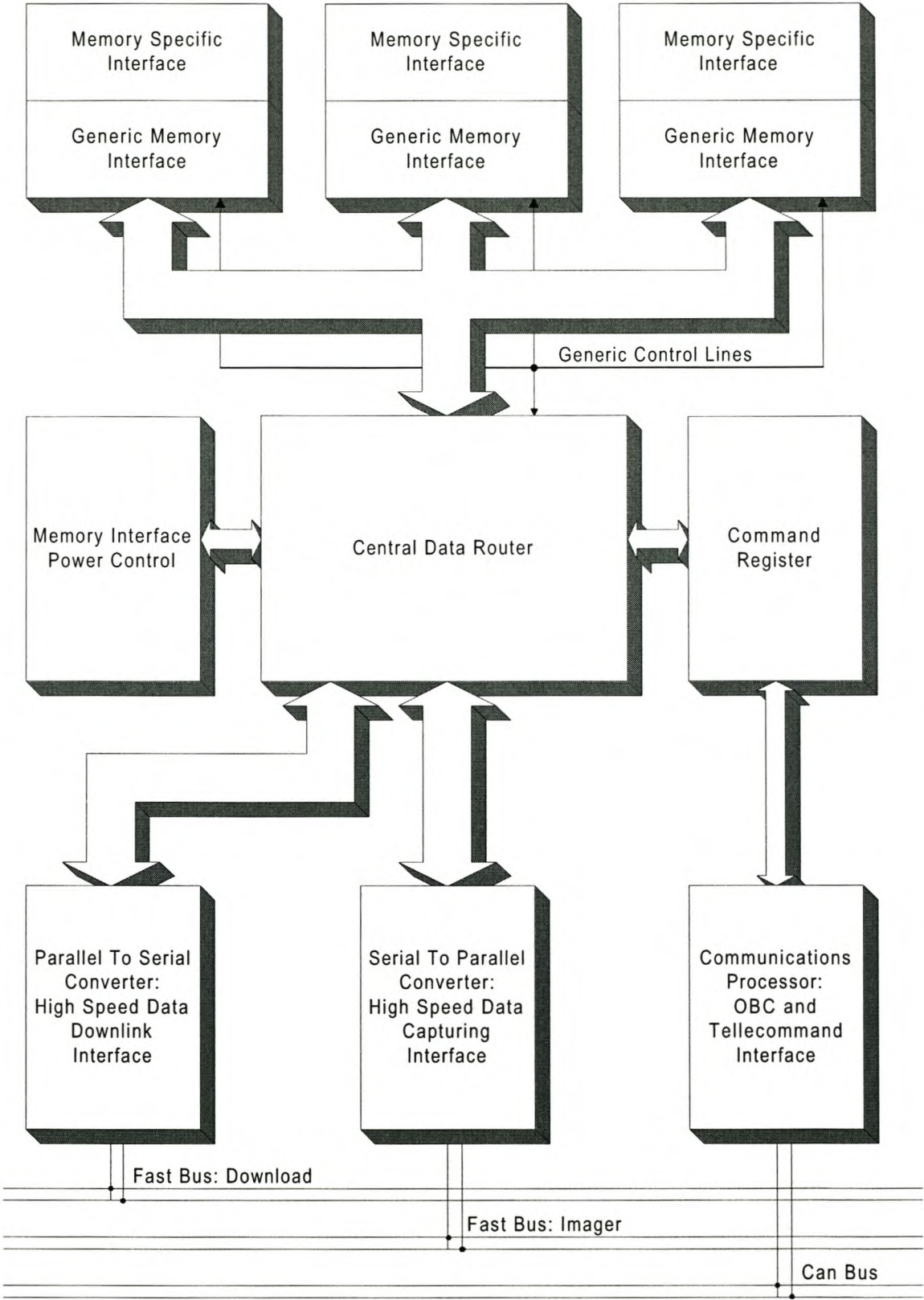


Figure 16 – Memory Drive Block Diagram

3.3.1 Central Data Router

The data router is the heart of the memory drive. This component will gather information and write it to memory as well as read data to be transmitted to other subsystems on the satellite. Here most of the functionality of the drive will be implemented.

In Figure 16, three memory blocks are shown. This is just for illustration, and does not imply that there are only three memory blocks. The modularity concept of the memory drive requires that any number (to a maximum of 255) of memory blocks can be hooked up to the data bus. One of the main functions of the data routing system will be to manage the data storage between these blocks.

This routing algorithm will have different modes, accumulating the data and storing it in memory. Another mode will be to serialise the incoming image data and store it sequentially in the memory.

3.3.2 Generic Memory Interface

The generic memory interface is a key concept in the new memory drive design. This will be a standard interface that connects to the data routers' main data bus to the memory. One GMI (generic memory interface) will exist for each block of memory.

The GMI will also have modes. The data router will set these when the interface is selected. Each mode will indicate whether the information to follow on the bus is data or an address, read or write operation, etc.

When an address is received it will be sent to the memory specific interface, which will in turn write the data at the correct address. Data will be passed directly to the MSI (memory specific interface), and in turn written to the memory. Read operations will occur in reverse.

These blocks of memory may be of any type; different memory types may even be mixed on one implementation of a memory drive. Only one type of memory can be used with one memory interface, because each GMI will have a memory specific interface that will cater for the specific needs of the memory. The interface between

the GMI and MSI will be standard; concerning read, write and address set-up protocols. Once the command is received by the MSI it writes/reads to/from the memory with that memory type's specific protocol, timing, etc.

3.3.3 Command Register

The command register is the heart of the automation of the memory drive. All the set-up information concerning the specific implementation of the memory drive will be stored here. The data router will then use the relevant information and configure itself accordingly. There will be static and variable parameters. The static parameter holds the size (in address width) of each memory block. This is needed to correctly resolve the address and select the correct block. Variable parameters will be used to set different operating modes. These parameters can be changed in real-time.

Variable Parameters:

- Discard every second line. This function is useful when a low-resolution image is to be taken. With every second line discarded the image will be smaller, but enough detail will be preserved to still have a recognisable image. This will typically be used to survey a possible high-resolution image site. The format of the image data is dependant on the imager, and can differ for different implementations of the memory drive. This function will have to be customised for each implementation.
- Skip memory blocks. If for some reason a block of memory needs to be skipped in a sequential write this will be indicated here.

This command register sub-system's main function is to create a defined interface to the memory drive. All the software and telecommand operations will take place via this interface. Instructions will be used to control the memory drive. These instructions will be written to the command register via the communications processor, which will in turn invoke the data router with the correct function. Any return data or information will also be stored in the command register, where the OBC can retrieve it.

This defined interface will simplify the software immensely, and also enable a person without detail knowledge of the memory drive to write software.

3.3.4 Memory Interface Power Control

This subsystem will control the power distribution to each GMI. When an address is set up via the command register the appropriate GMI is automatically switched on. Each GMI can also be switched on or off manually via an instruction written to the command interface. Note that the data router does not know when an interface will not be needed anymore, thus all GMI's have to be switched off manually by the controlling software.

3.3.5 Serial To Parallel Converter

The serial to parallel converter is the most speed sensitive device on the memory tray. Here the data from the imager is converted from its serial form into 32 bit data words. The first bit to arrive is stored in the least significant position. These 32 bit words are then stored at the currently active GMI's address.

3.3.6 Parallel To Serial Converter

When downloading data to earth a high-speed serial link will be used to transfer the data from the memory drive to the appropriate RF subsystem. When enabled this converter automatically reads data at it's own pace from the current memory location in the active GMI. Its data transfer rate is determined by the system to which data is being downloaded. Once again the LSB is placed on the serial bus first, thus the bit stream is sent out in the same order as it came in.

Chapter 4

DETAIL DESIGN

4.1 Introduction

Using FPGA's to implement the memory drive is certainly a versatile and cost effective method. SRAM based devices can be used to build and test a prototype design, which will drastically bring down development time and expenditures. SRAM based FPGA's are volatile, and have to be re-configured every time power is applied. This will be very useful when debugging the system, because any errors detected can be fixed via software and a new configuration file created. No hardware redesign will be necessary.

An engineering model can be created with SRAM based devices, and when all the bugs have been corrected the final version can be implemented in non-volatile, or fuse-link devices. The only difference will be timing constraints, which will have to be checked whenever the design is ported between device types, or manufacturers. This does not present a problem, seeing as test procedures can be developed using the prototype, and these procedures can then be used to qualify other implementations.

The design of the memory drive will be done using a hardware description language. The functionality will be described with VHDL, which will be the core design. The VHDL code is written with a broad spectrum of target devices in mind. This will not be foolproof though, as different FPGA's have different architectures, and porting from one technology to the next may require modification. The core design will stay the same though.

Altera manufactures FPGA's, and have a number of different programmable products. Their software development platform Max+Plus II is available to students from the University of Stellenbosch. This platform was used to develop and simulate the VHDL code.

4.2 Design Structure

One important lesson that was learned from working on the SUNSAT RAMDISK, is that a structured design approach must be followed. This is especially important if the design will be expanded upon, tested or maintained by other people. A structured approach also promotes reliability, a very important topic for any system that will operate onboard a spacecraft.

Debugging is another important factor to be considered. With a structured design set procedures can be followed, and the problem located easily and efficiently.

The design methodology that is used with the memory drive is state machine based. As far as possible a state machine approach will be followed. Another very attractive attribute of state machines is that extra functionality can be added without any major changes to the design structure. Typically only an extra state, or states are added to perform the required function.

To complement the state machine approach the design will also be kept synchronous as far as possible. Some signal edges will have to be used, but most assignments will be made on the rising flank of the system clock. This promotes stability, and ensures a clean, or glitch free design.

4.3 Design Tradeoffs and Choices

4.3.1 Drive Structure

It is possible to have one large block of memory, which would simplify the structure. The disadvantages include a limited application potential, and a non-redundant design. The modular structure using generic memory interfaces discussed above was chosen due to the following reasons:

- Versatile – Any type of memory can be used without changing the structure of the Memory Drive. The memory capacity is also easily variable depending on the required application.

- Redundant – If one memory module fails normal operation can be continued by simply skipping the affected module. The data router will implement this function.
- Simplified maintenance – Memory failures can easily be isolated to a specific memory module, and the appropriate steps can then be taken.

4.3.1.1 Data Bus Structure

The focus of the Memory Drive is on a modular design. The data bus structure must support any number of memory modules (with a maximum limit of 255). A standard structure, which is not dependant on the number of memory modules, must be used.

Only one memory module may have control of the data bus at any given time, thus an activation scheme for each module is necessary:

- A chip select (CS) line – This will need one line for each memory module, and is not desirable.
- A unique address for each module – This is more desirable, but will require more than one write cycle to facilitate single control of the bus.
- A unique address with a control line – A combination of a unique address, or ID, with a control line will serve as a good solution. The control line can be used to indicate to the other GMI's not to listen on the bus for their ID, which will save power. This will be due to the inhibition of the clock driven listening scheme. And because the CMOS device is not running at a high frequency it will dissipate less power.

4.3.1.1.1 Data Bus Width

As already mentioned a 32 bit data bus is used. This was chosen due to a number of reasons which will be discussed below.

Advantages of a wide bus:

- Speed – A wider bus implies that more bits can be written or read per cycle. This will increase the total data transfer rate.
- Lower clock frequency – To achieve the same data transfer rate with a wider bus a slower clock frequency is needed. This has many advantages: slower devices can be used, which will cost less. Power consumption will be less, because with FPGA's (CMOS technology) a higher clock frequency dissipates more power. Timing constraints of the system are more easily met.
- Less set-up cycles – When setting up an address fewer cycles will be needed to transfer the whole address.

Disadvantages of a wide bus:

- Size – A wider bus obviously requires more board space, which increases weight, makes the layout more complex and board manufacturing costs are also more expensive.
- Resource usage – More resources such as I/O pins are needed. When storing or buffering any data in the control logic more cells are necessary. This will lead to larger, and more expensive devices.
- Power – With FPGA's power consumption increases if more I/O pins are used [1].

With a 32 bit structure it was found that the speed requirement for the memory drive could be met by using a system clock of 5 MHz, which increases reliability by being more resistant to noise and interference from other systems. If an 8 bit structure was used the minimum clock speed would have been 20 MHz.

With 32 bits a 4 Gbyte memory space can be addressed. This was chosen as the maximum size for a GMI and implies that only one cycle is necessary to set-up an offset address in a GMI.

The memory drive was found to fit in middle of the range FPGA devices, and the largest FPGA used has 208 I/O pins.

4.4 Altera Implementation

Figure 17 is the top-level graphics definition file (.GDF) for the design. Here all the functional blocks are joined together, and proper integration can be verified by simulation of this structure. This is the implementation of the block diagram depicted in Figure 16. The full set of VHDL code for this design can be found in Appendix B.

MEMINT0 and MEMINT1 depicts the generic memory interfaces, or GMI's. There can be any number of these interfaces up to a maximum of 255, depending on the required amount of memory for a specific implementation. Each interface will be assigned a unique identification number and will be in a sleep mode unless activated with the correct ID. When active read/write operations can then be performed with this interface.

The ROUTER block is the heart of the memory drive. Here the data which is to be stored will be collected and sent to the correct memory location. Other tasks such as address set-up and memory management will also take place here.

The command register is contained in the ROUTER block. This will define the command and control interface to the memory drive. All operation will have a command code, which will be stored in a stack in this block. The ROUTER will then perform these commands.

POWERCTRL controls the GMI power. A GMI is switched on or off by writing its ID to this module with a control bit which specifies whether to switch it on or off.

SERTOPARALLEL converts the incoming serial data stream into 32 bit words. These are then routed to the active GMI by the data router, where it is stored in memory. The first bit to arrive is the LSB.

PARTOSERIAL serialises data read from the currently active GMI. This module automatically generates the signals that initiate a read cycle. The speed at which data is read from memory depends on the clock it uses to serialise the data. This clock will depend on the speed at which the data is to be downloaded. The LSB is shifted out first.

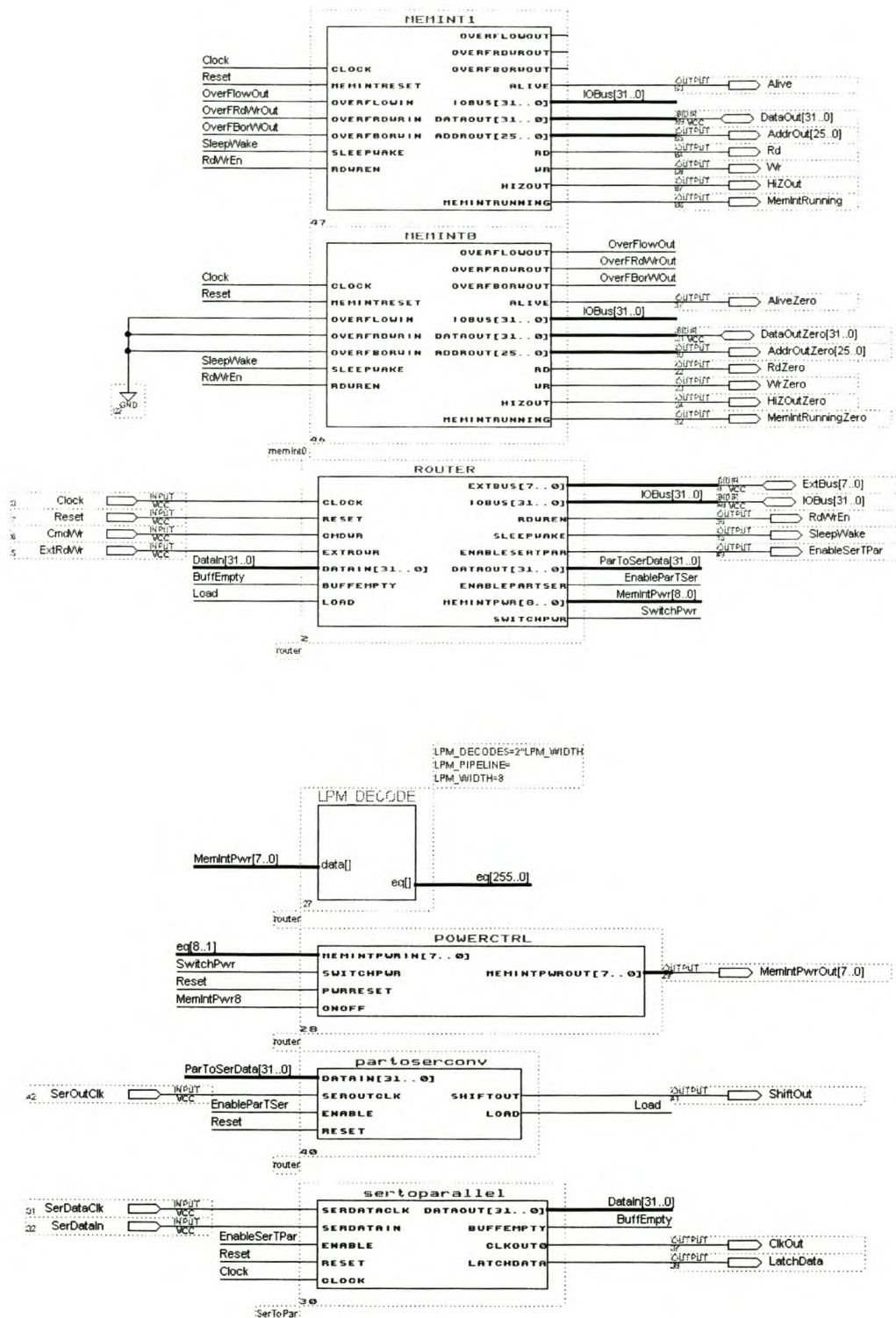


Figure 17 – Memory Drive Implementation in Max+Plus II

4.4.1 Generic Memory Interface

Figure 18 depicts a block diagram for the generic memory interface. The GMI is built around a state machine. This state machine is enabled by the GMI ID Monitor, and controls what happens to the incoming data on the 'IO Bus'. The process of each block relates to the VHDL process, which implements that function.

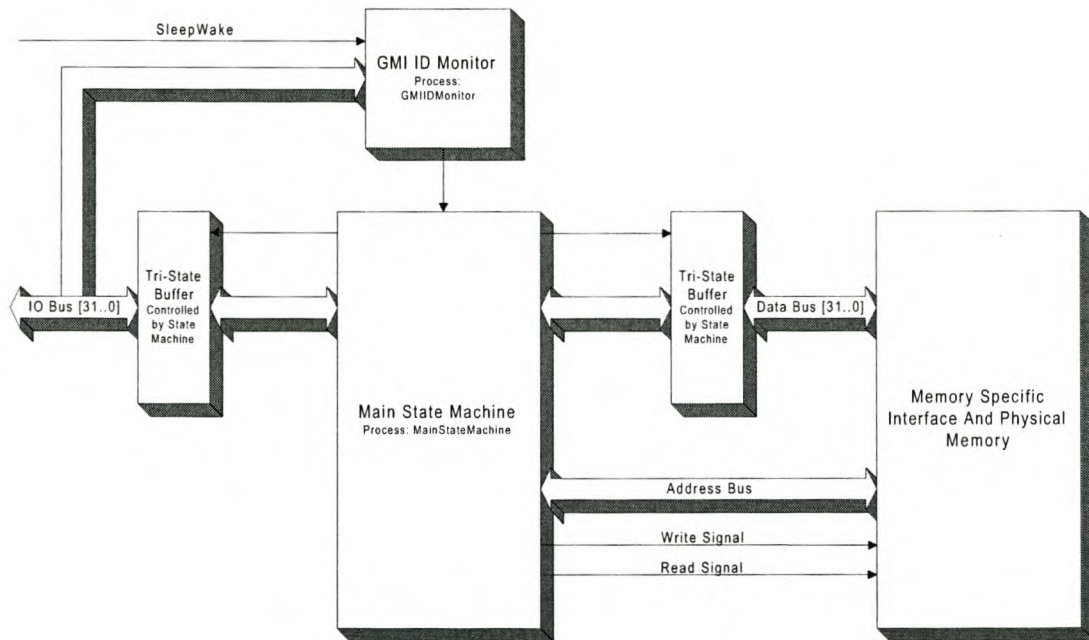


Figure 18 – Generic Memory Interface Block Diagram

4.4.1.1 GMI Selection Process

The challenge is to implement the memory interfaces in such a manner that a minimum number of changes or customisation of each interface needs to be performed. It is highly undesirable to have unique signals travelling to each interface. The reason for this is the modular focus of the memory drive. Interfaces need to be inserted and taken away without major impact on the rest of the design. The same board layout, software and firmware must still be valid.

The above statements require that a standard hardware interface is necessary for each interface. As already mentioned this can be achieved by assigning a unique identification tag. This tag is then coded in the interface's VHDL. The possibility to

change this tag in real time must also be investigated. This will be useful if any of the interfaces, or their memory become inoperable. The tags can then be changed to reflect a flat memory space.

Because the interface will be connected to the central routing device via a bus structure a method must be found to ensure that only one device is active on the bus at any given time. This selection process must be very robust because a fault may lead to two devices driving the bus at the same time, which can lead to device failure.

The obvious choice is to have a chip select line for each interface. But referring to the discussion in section 4.3.1.1, this is not desirable. If a large amount of interfaces are used, precious IO lines on the routing device will be lost, and the board layout will also become more complex. The solution used is an awake/sleep signal (designated 'SleepWake'), which is connected to all the interfaces. If the signal is a logic one, the interface will listen on the bus for their ID.

The central routing device acts as a bus master and assumes control after power-up, or a reset. Each interface listens on the bus for its ID, and upon receiving it becomes active. As soon as 'SleepWake' goes high again the internal state machine resets. The other interfaces, which did not receive their ID's, are now in a low power mode, until another 'SleepWake' event. This low power mode is entered by inhibiting the clock.

4.4.1.2 Address Structure

An interfaces' ID is directly related to its absolute address. The address can be broken up into a segment and an offset, similar to what is used in the 8086 processor architecture. The offset is the address inside each GMI, and the segment indicates which GMI to use. The static variable described in section 3.3.3 indicates the width in bits of each GMI. This is also the width of the offset address.

The remaining bits in the absolute address makes up the GMI's ID. This cannot exceed 255, as the GMI ID monitor (depicted in Figure 18) only checks one byte for the ID. It was not deemed necessary to make provision for more than 256 GMI's.

4.4.1.3 GMI State Machine

In an effort to minimize the number of control lines needed a memory interface is controlled by modes. These modes indicate the following:

- OBC data read or write – For OBC data transfers 8 bits are used (the remaining 24 bits are ignored). After each read or write the address is automatically incremented by one.
- Data capture or dump – These are 32 bit operations. Here the address is incremented by four after each cycle.
- Address read or write – The address must be set-up before the first data access operation. For subsequent operations the address is incremented automatically.
- Status read or Control write

A single control line is then used to generate the read and write pulses. The data on the bus is then interpreted according to which mode is set. The mode is bundled with the ID tag, thus each time an interface is activated it is set up to perform one function. If another function needs to be performed the interface is reset (by driving 'SleepWake' high) and re-selected with the new mode.

In 4.4.1.2 it was mentioned that the GMI ID monitor checks one byte for the correct ID; this is the LSB. The second byte contains the operating mode for the GMI. This mode is sent to the state machine, which in turn performs the correct function.

4.4.1.4 Interface to the MSI

Figure 18 also contains a memory specific interface. This was not implemented, and will depend on the type of physical memory used. The interface to the MSI consists of a 32 bit data bus, an address bus (the width is set via a static variable), a read and a write signal.

The operation is synchronous: depending on whether read or write is asserted; on the rising edge of the system clock the read/write must occur at the address on the address bus. This simplifies the implementation of a MSI, which implies that any type of memory can be used with the memory drive.

4.4.1.5 Spanning Multiple GMI's

Another very important aspect of the memory drive is to create a continuous, or uninterrupted memory space. When writing a large block of data – typically an image – there is not enough time to have the data router switch between memory interfaces. This switching must happen automatically, but with a certain degree of intelligence. In normal circumstances each block must transfer control of the bus to the next, or adjacent memory interface without dropping any data.

As discussed above unique control lines to each interface is undesirable, thus control must be passed from one interface to the next without intervention by a third party. This is not a problem, because all the interfaces can be "daisy chained" together. Figure 19 illustrates this chain. This line will then pass control from one interface to the next.

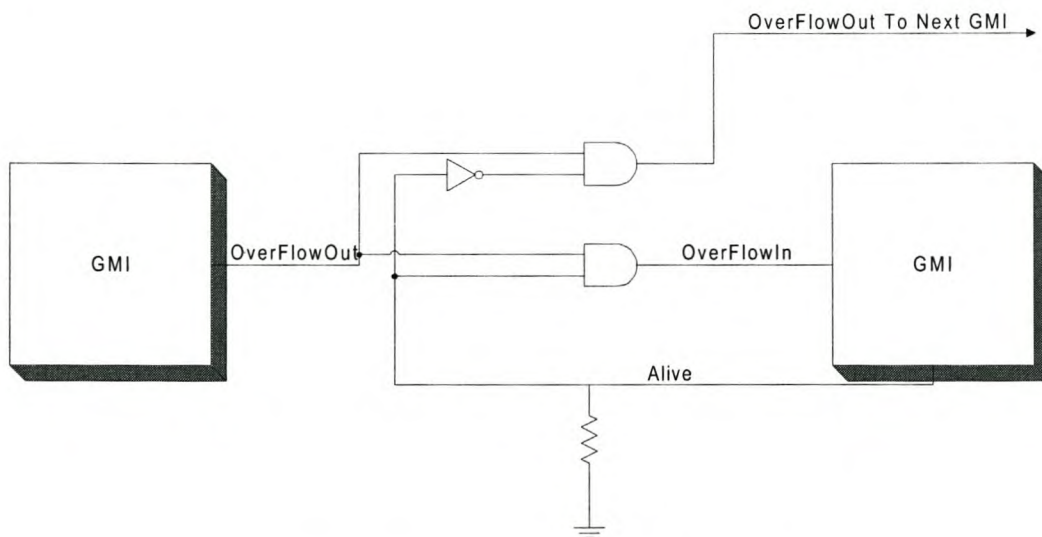


Figure 19 – OverFlow: Passing Control From One GMI to the Next

The problem arises when an interface must be skipped. This could be due to any number of reasons e.g. memory failure on that interface, or protected data, etc. This is where the intelligence switching must be used. Seeing as a third party controller has been ruled out the memory interface itself must indicate whether it is to be skipped or not. This mechanism must also skip an interface automatically if the interface has failed.

This is achieved by inserting a multiplexer in the daisy chain. The select line comes from the target interface. If this select line is low the interface is skipped, and with a high it is included. This select line will be called 'Alive'. A pull-down resistor will be used to ensure that the line stays low in case of an interface failure, which will always cause it to be skipped.

4.4.1.6 External Buffers

The IO bus is connected to each GMI, as well as the data router. A problem observed in the SUNSAT RAMDISK was the driving of dead logic. This was caused by an active bus connected to inactive devices.

Here this problem will be overcome by adding an external bi-directional buffer between each GMI and the IO bus. These will be powered from the same power bus that drives the central data router. The buffer enable line will be connected to the GMI's power line. A switched off GMI will thus not load the bus. Note that not only the bus will be connected through an external buffer, but all the control lines too.

4.4.2 Central Data Router

The central router is the heart of the system. It gathers data and according to predefined commands, decides where to send it. Figure 20 is a block diagram representation of the data router and command register. These two are depicted together because essentially the data router is only one state machine, and is controlled by the command register.

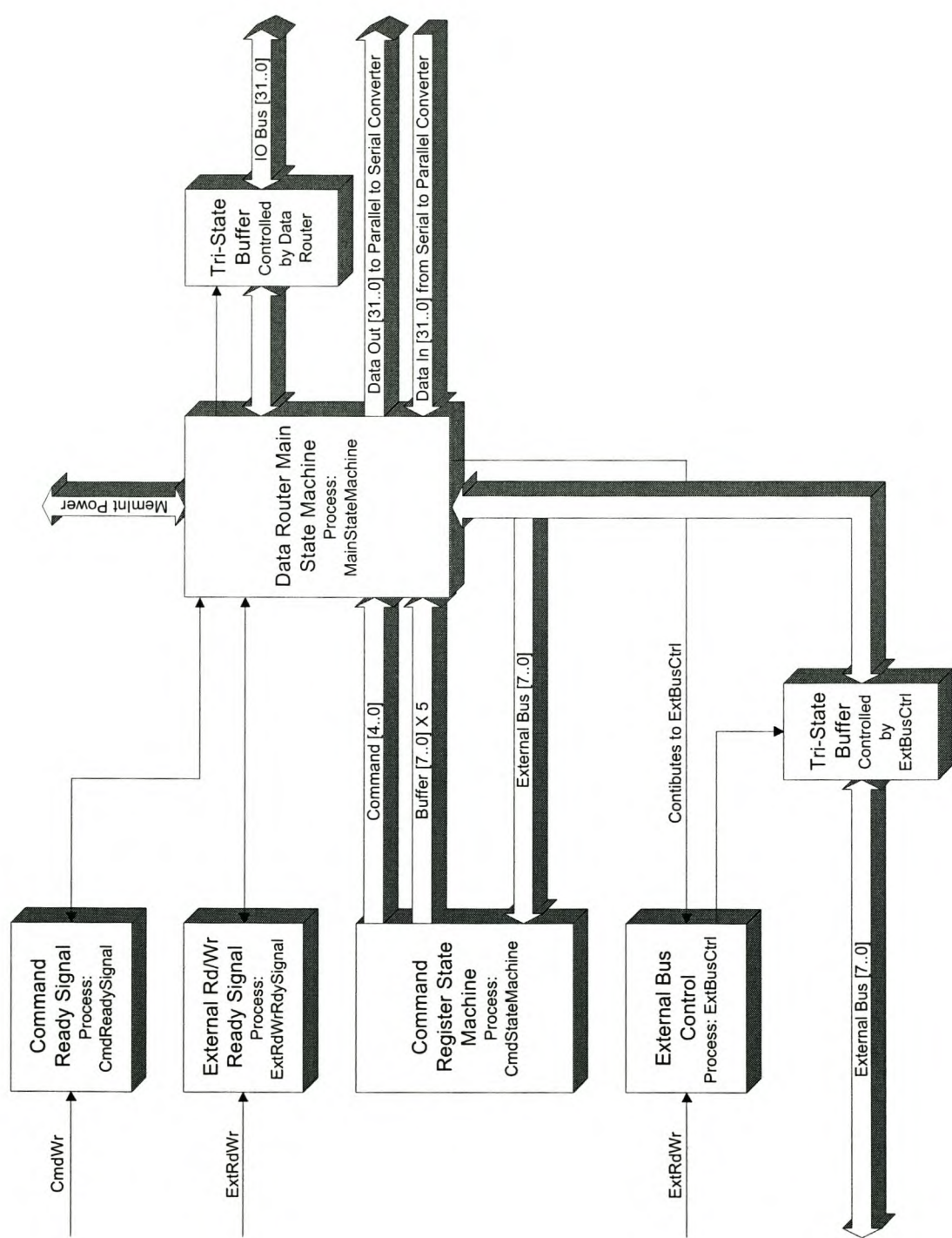


Figure 20 – Central Data Router and Command Register Block Diagram

4.4.2.1 Data Paths

There are four main data paths:

- **IO Bus** – This 32 bit bus connects all the GMI's on the memory drive to the data router. It is bi-directional and the data router acts as bus-master. If set in the correct mode by the router the GMI's can also drive this bus. This is a general-purpose bus and transfers data, addresses and control information such as GMI ID's bundled with commands to set the mode for that GMI.
- **External Bus** – Here is the connection to the outside world. This is an 8 bit bi-directional bus, which connects to a communications processor and ultimately via a serial CAN bus to the OBC. The data router only writes data to the external bus if an OBC read command is set. The command register monitors this bus for commands and when a write instruction is received the bus is routed to the lower byte of the IO bus, which is in turn sent to the previously set-up GMI.
- **Data In** – High-speed data will arrive in serial form from a data-generating device such as an imager. The serial to parallel converter breaks the input stream up into 32 bit words, which are sent to the data router via this bus.
- **Data Out** – 32 bit words read from the current location in memory is sent to the parallel to serial converter via this bus. The resulting serial stream is sent to a communications subsystem where it is downloaded to a ground station.

Data width will vary from implementation to implementation. For example an imager with 10 bit resolution will have 10 bit data packets. The memory stores data in bytes and to accomplish this the packets will be broken up. Effectively a serial data stream will be stored in memory. This splitting up of data packets presents a reliability problem. If synchronisation is lost all the data will be rendered useless. This situation will be very rare though, because a synchronisation loss will only occur due to bits being dropped by the data router. If this happens the data router has most probably failed catastrophically, in which case a redundancy solution will have to be used. If the

error can be recovered the best solution will be to restart the memory read operation from the beginning.

At start-up, or after a reset the router will have control of the data bus. As mentioned above all the memory interfaces are listening for its ID. Using the command register (which will be discussed in 4.4.3) to obtain the start address of a memory operation the ID of the correct memory interface is placed on the bus for one clock cycle. On the rising edge of the clock the memory interfaces samples the bus and activates if the correct ID was found. Refer to 4.4.1. The required function, or mode is also sent with the ID. This tells the memory interface which type of operation it must perform.

4.4.2.2 Data Router State Machine

The state machine follows the same pattern for all operations. The only differences between instructions are which signals are asserted, and which data paths are connected together.

Once an address has been set-up the correct GMI's ID is stored. Until a new address set-up occurs all GMI selections are done with this ID. Each new instruction goes through a GMI selection state. Even if the correct GMI is still selected from the previous operation this reselection is necessary, because the mode is bundled with the selection packet (byte 0 = GMI ID; byte 1 = mode)

Only one command sequence will be illustrated with a flow diagram, the rest will follow the same pattern. Simulations of these commands are documented in Chapter 5.

Figure 21 shows a flow diagram of a memory read cycle. This is one of the commands that the data router can perform. See 4.4.3.1 for all the commands. The signal 'ExtRdWr' is an incoming line from the communications processor, which originates from the OBC. 'RdWrEN' is the read/write enable line for the GMI's and when high, a read/write operation will occur at the next rising clock edge.

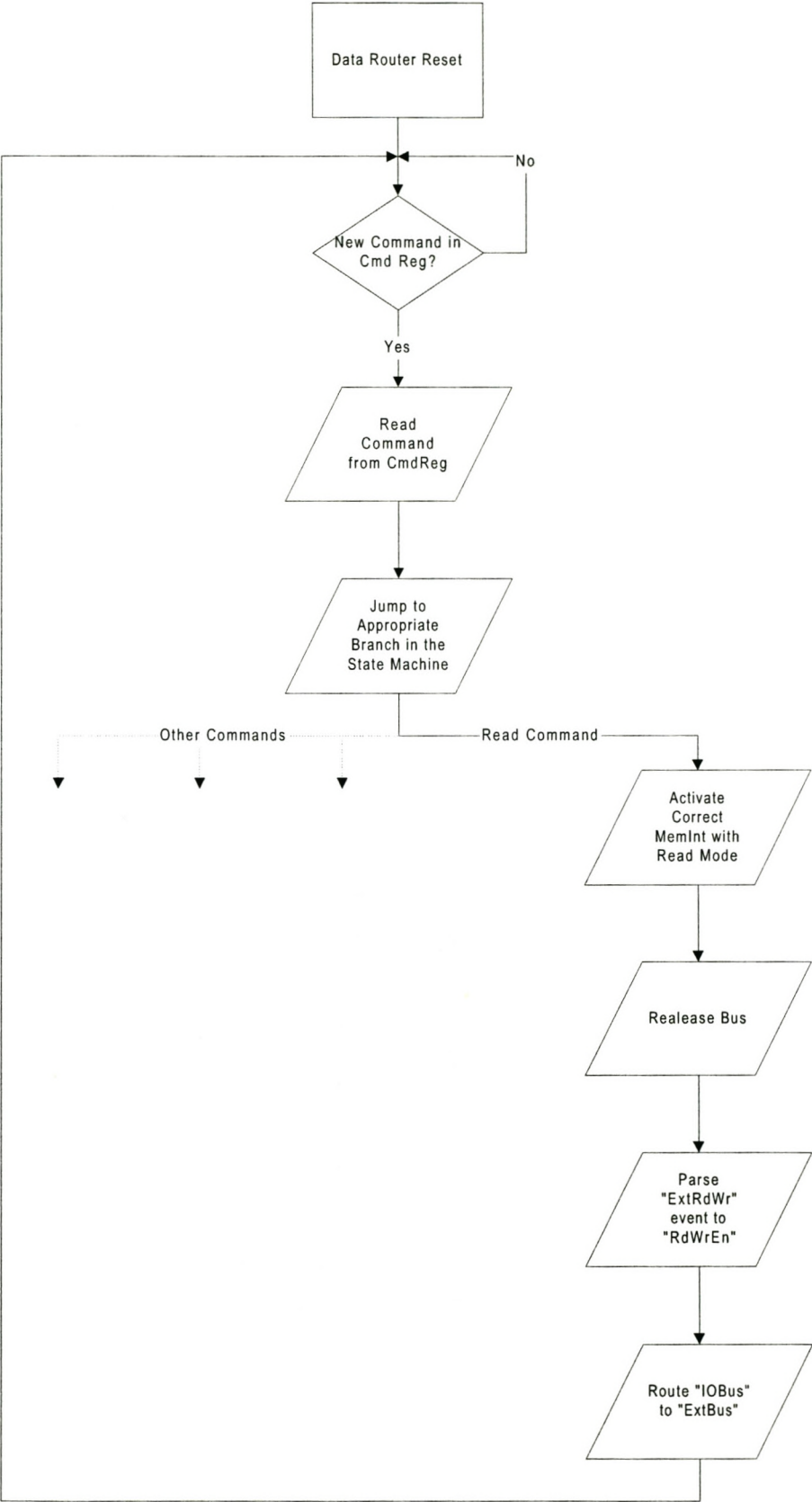


Figure 21 – Data Router OBC Read Cycle

4.4.2.3 Power Control

Power dissipation is an important topic due to the nature of a satellite. If memory is not in use, the interface controlling it must also not dissipate any power. Normally the switching on/off of components is controlled via telecommands, though in this case the data router is in the best position to decide which interfaces should be active. The router will not know when information will not be needed anymore, thus switching off of GMI's will have to be managed by the OBC.

One power control line will be necessary for each interface. Once again due to the modular nature of the memory drive the exact number of memory interface will be determined by the specific application. If there are enough IO lines available (one for each interface) on the data router device a direct connection to the power switches can be made. If more lines are needed a decoder is used. A minor modification to the VHDL code will have to be made when using a decoder.

4.4.3 Command Register

A block diagram of the command register is available in Figure 20. All the blocks on the left of the central data router state machine block belong to the command register.

The command register will act as a type of software stack. This will be the command and control interface to the memory drive. The data router has a number of defined operations it can perform: e.g. setting up a start address for in preparation for a data capture, etc. These operations will each have a unique code. This aspect of the memory drive will approximate a software processor.

The controlling device (an on board computer (OBC)) will write the data router operation codes into the command register. As soon as new instructions arrive the command register will inform the data router, which will take appropriate action. The result of that action (if appropriate) will be written to the command register by the data router. This can then be read by the OBC.

This defined interface will simplify the software driver needed to control the memory drive. The software engineer need not have intimate hardware knowledge of the memory drive, which was a problem with the SUNSAT RAMDISK.

In this design an eight bit data bus between the OBC and memory drive was assumed. This may change depending on each implementation, but the command register concept will stay the same.

4.4.3.1 Commands

The commands are stored in a table in the command register. The incoming instruction is compared to this table and subsequently the data router can perform the correct operation.

Each command consists of a byte, followed by up to five bytes depending on the specific command. The command byte contains two fields:

- **Command Length:** This length is contained in the three most significant bits, and indicates the amount of bytes to follow.
- **Actual Command:** This consists of the least significant five bits and indicates the type of operation to be performed.

List of commands:

- **Write Data to Memory:** If the OBC needs to write data to the memory drive this command is used. 'ExtRdWr' will indicate when a new byte is available for writing. This sequence will be repeated until another command is placed in the command register. Note – with both "Write Data to Memory", and "Read Data to OBC" the memory drive is assumed to process instructions faster than the OBC. This means that the OBC can write or read data to/from the buffer sequentially at full speed, and there will never be a buffer over – or under run.
- **Read Data to OBC:** Data read from memory by the data router will be written to a buffer in the command register. The OBC can now in turn read this data from the buffer. The

command register will signal the data router when the buffer is vacant, which will then read the next byte in memory and place it in the buffer. This will continue until another command is written to the command register.

- **Set-up Address:** This command will be accompanied by the address, which is to be set up in the memory. The data router will then read this address and perform the necessary functions to set-up where the next write or read operation will take place.
- **Write GMI Command** This function was added to write any specific set-up or command information to the GMI. The use of this will depend on how the MSI is implemented.
- **Read Status from GMI** As above this function was also added to deal with specific MSI status information.
- **Capture Data Block:** Here the data router is instructed to start capturing data on the high-speed interface. This will typically be raw image data from the imager, or data uploaded on a high-speed uplink. This command implies that the start address has already been set-up. The size of the block of data to be captured is placed in a dedicated buffer in the command register. The data router will read this buffer and subsequently only capture the correct amount of data.
- **Dump Data Block:** This instruction indicates that data is to be read from the memory and sent to a high-speed downlink. As above this command also assumes that the address has already been set-up. An amount of data to dump will not be

specified, as the process can be stopped by writing a new command.

- **Force GMI On/Off** These commands provide the functionality to manually switch a GMI on or off. A buffer is used to indicate which interface is to be switched

See Table 2 for a list of command codes. Note that space has been left for extra commands to be added. With the state machine approach new command will be very easy to implement: the command register will not change at all, and states must be added to the data router state machine.

Command	Command Label	Command Code	Bytes to Follow
Write Data to Memory	CwrData	00000000	0
Read Data to OBC	CrData	00000001	0
Set-up Address	CsetAddr	10100010	5
Write Command to GMI	CwrCmd	00100100	1
Read Status from GMI	CrStatus	00000101	0
Capture Data Block	CcapData	10100110	5
Dump Data Block	CdumpData	00000111	0
Force GMI On	CforceOn	00101000	1
Force GMI Off	CforceOff	00101001	1

Table 2 – Command Code Table

4.4.4 Serial to Parallel Converter

Figure 22 shows the serial to parallel converter. This system is divided into two subsystems. The SHIFTRREG block is the shift register, which converts the serial data to parallel. The resulting word is now stored in a circular buffer (CIRCBUFF).

To save power this system has an 'Enable' signal. The data router generates this signal. When a data capture operation is performed 'Enable' is asserted, and the serial to parallel converter operates normally, otherwise the clocks are inhibited, which saves power.

4.4.4.1 Circular Buffer

This system buffers the incoming data from the imager, or data-generating device. The circular buffer consists of four 32 bit storage elements, a head and a tail counter. Each new word is written at the head address, and the head is then incremented. Data is removed from the buffer by reading the word at the tail address and incrementing the tail. 'BuffEmpty' indicates when the buffer is empty. This signal is asserted when the head and tail do not point to the same location, which means that data is ready for collection.

This load of data is a synchronous operation; if 'LatchData' is asserted the word on the bus is stored at the next clock event.

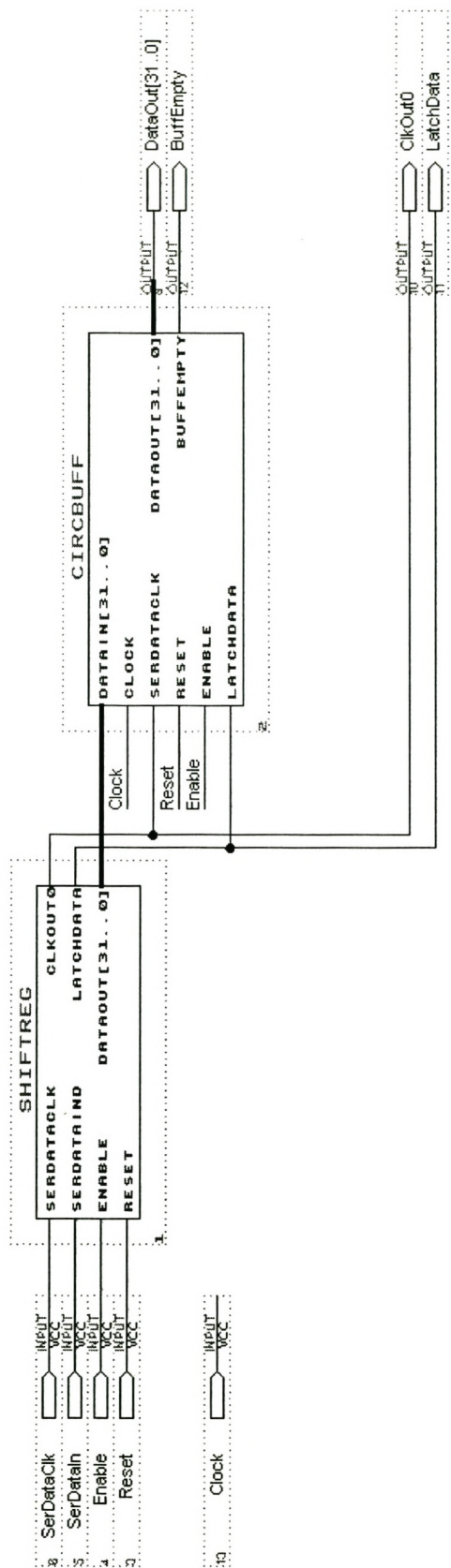


Figure 22 – Serial to Parallel Converter

4.4.4.2 Shift Register

Design of the shift register presented a speed problem. At the required data rate a 130 MHz clock must be used to latch each bits in the serial input stream. This gives a clock cycle time of 7.7 ns.

There are shift registers on the market that can shift data at these speeds, but they are expensive. If a normal FLEX6000 FPGA from ALTERA is used to implement a shift register simulation shows that the average delay to shift one bit is 8 ns. This delay will cause bits to be dropped at 130 MHz.

The solution used is shown in Figure 24. Four shift registers are used to shift the incoming data. This is done by splitting the clock into four (see Figure 23), and using each of these to drive the shift registers. Each shift register only needs to operate at a quarter of the frequency. The resulting four busses must now be interleaved to obtain the correct bit order.

The four clock edges are delayed from the input clock edge. To ensure that the data is still synchronised to the new clock the serial input stream is put through a latch. A clique is formed with the clock splitting circuit and this latch. In Max+PlusII a clique specifies that these components must physically sit close together in the FPGA. This ensures that the delay through them will be roughly the same.

A FLEX6000 device was tested (see Chapter 5) and found to work at the required frequency. The EPF6010ATC100-1 was used in the simulation.

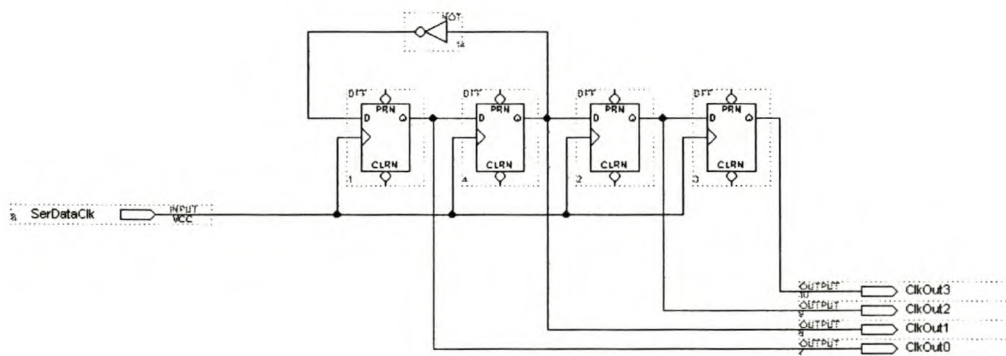


Figure 23 – Clock Splitting Process

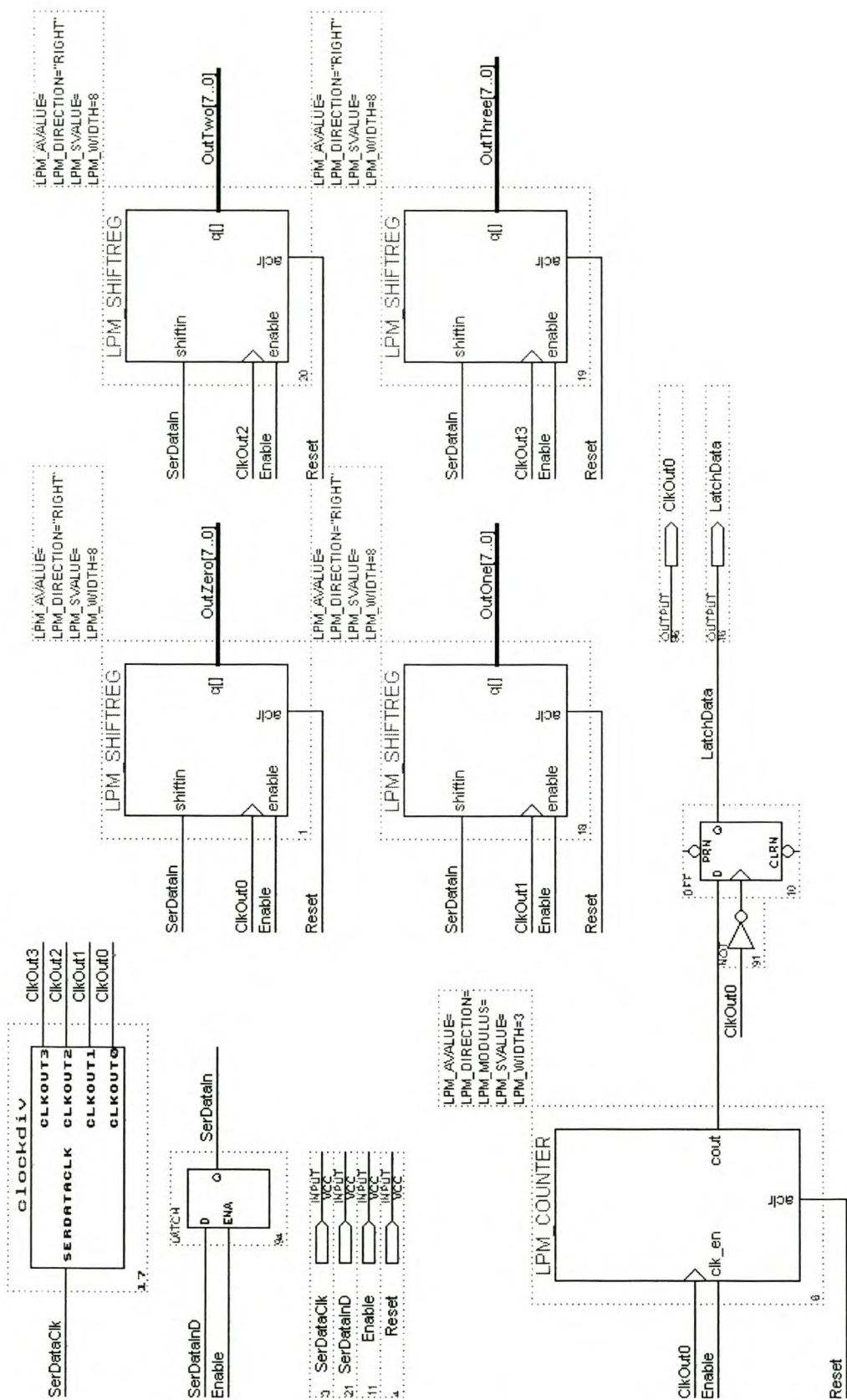


Figure 24 – Shift Register

4.4.5 Parallel to Serial Converter

Figure 25 shows the parallel to serial converter. This is a strait forward serialiser with a 32 bit input bus. The LSB is shifted out first. 'Load' signals the data router when to make a new word available.

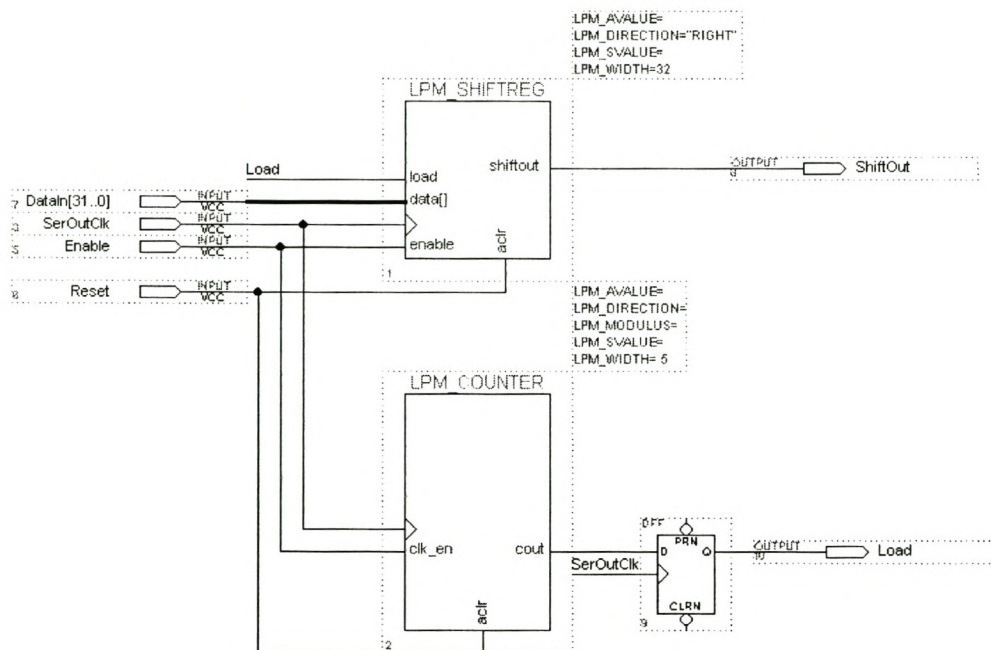


Figure 25 – Parallel to Serial Converter

Chapter 5

DETAIL SIMULATION

5.1 Introduction

In this chapter the actual design is verified. For this project the design will not be realised in hardware, which means that simulation must be heavily relied on. This is in general very good practice, seeing as a properly simulated design normally only needs to go through one hardware iteration.

This of course saves money and time. The more time spent verifying a design properly the less time needed to debug hardware. Emphasis must be placed on simulating a design thoroughly, as faults can easily be overlooked. In a simulation environment the state of all inputs are defined by the person verifying a project. If all possible situations are not recreated faults might be overlooked.

An especially complex situation arises when simulating a design containing a common bus structure. When putting an input bus in a high impedance state the simulator does not recognise it as high or low, and when this bus is read by the device under simulation an undefined result occurs. It is often necessary to bypass this problem by forcing a result. When doing this the bus is assumed to be in some state. One must carefully analyse whether the result is indeed a correct assumption.

5.2 Platform

Altera's Max+PlusII was used to simulate the memory drive design. This is a very useful tool, as functional as well as timing analysis can be performed which will highlight most errors in the design. Inputs can be manipulated at will and worst case timing requirements can be checked.

As mentioned above care must be taken to ensure that the correct result is obtained.

5.3 Simulations

The data router and command register will first be verified to work correctly. Then these systems will be connected to a generic memory interface as depicted in Figure 17. The serial to parallel converter and parallel to serial converter will also be simulated.

Please refer to Appendix B. This contains the VHDL code used for these simulations. A description of inputs, outputs and signals for the memory drive is available in 0. Refer to this if a signal function is not clear in the simulations.

5.3.1 Data Router and Command Register Simulation

Figure 26 is the simulation for the data router and command register. This simulation contains address set-up, OBC write and OBC read cycles. Each will be explained individually.

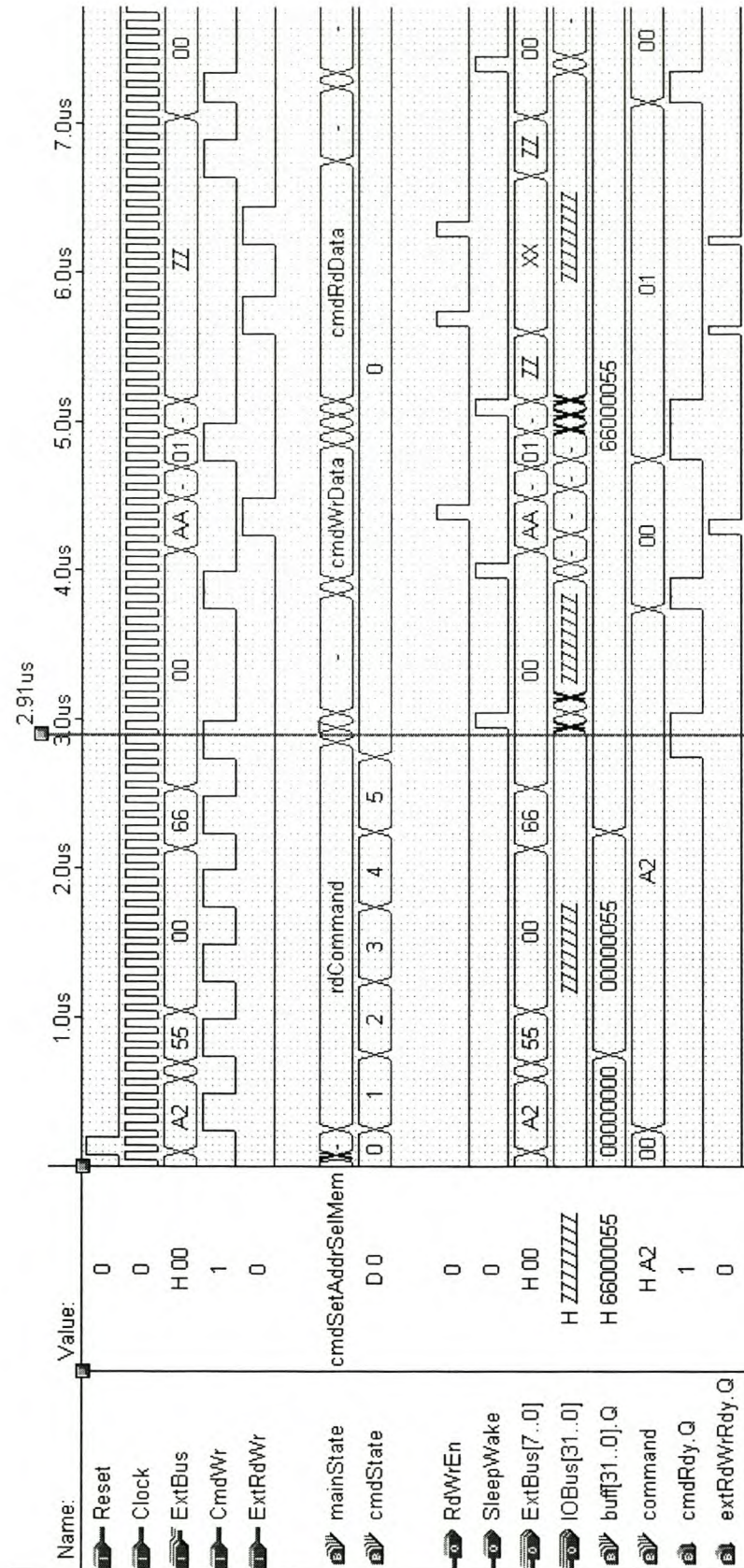


Figure 26 – Central Data Router and Command Register Simulation

5.3.1.1 Address Set-Up Cycle

'ExtBus' is the external bus from the OBC or telcommand system. This is the interface between the command register and the rest of the satellite. Control signals accompanying this bus is 'CmdWr' and 'ExtRdWr'. 'CmdWr' is used to indicate that the data on the external bus is a command, and 'ExtrdWr' indicates a data read or write operation.

The command for an address set-up cycle is 10100010B. The list of commands is available in Table 2. Referring to 4.4.3.1, the MSB three bits indicate the amount of data to follow, and the LSB five bits indicate an address set-up. This is clocked in and the address now follows, least significant byte first. The address can be five bytes long, which means 1000 Gbytes can be addressed. The address is now clocked in LSB first: 0066000055H.

Upon the fifth byte being received 'cmdRdy' signals the data router that an address is ready to be set up. 'CmemIntWidth' is a constant defined in the command register. This is the static variable mentioned in 3.3.3. This constant indicates how big each memory interface is. For this simulation it was set to 64 Megabytes, or 26 bits. The data router now determines the GMI ID and offset. The offset is the least significant 'CmemIntWidth' (26), which is 02000055H. The remaining bits makes up the ID, which is 19. Figure 27 shows how 'SleepWake' is asserted with the ID and the mode (CsetAddr = 02H) on the IO bus (Figure 27 is an enlarged section of Figure 26). Directly after the ID comes the offset address. Both will be read by the correct GMI on the rising edge of the clock. Note that the main state returns to 'rdCommand', ready for another command.

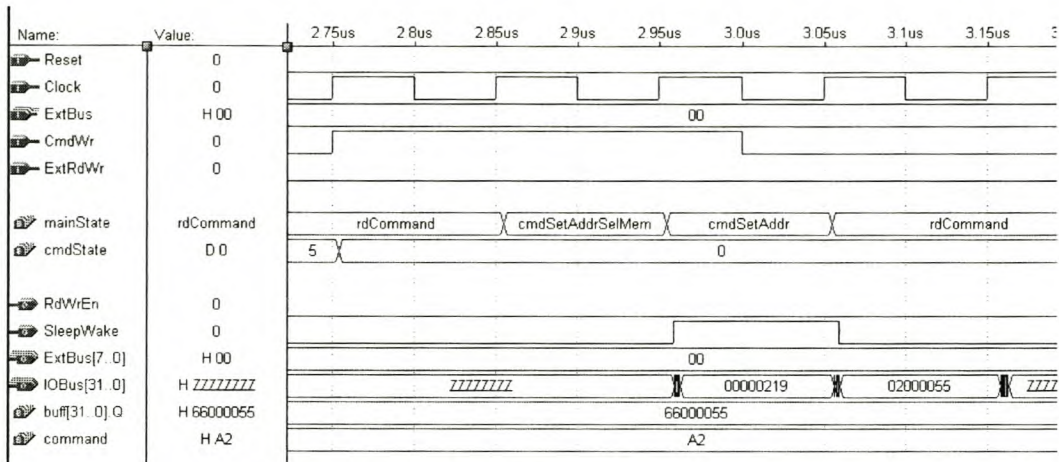


Figure 27 – Router Address Set-up

5.3.1.2 OBC Write Cycle

Figure 28 shows how a write command is executed. With 00000000B on the 'ExtBus' a 'CmdWr' is asserted. At the next clock edge the router state machine changes to 'cmdWrDataSelMem', and the ID with mode (CwrData = 00H) is put on the IO bus.

Directly afterwards AAH is clocked in by 'ExtRdWr', and 'RdWrEn' is asserted.

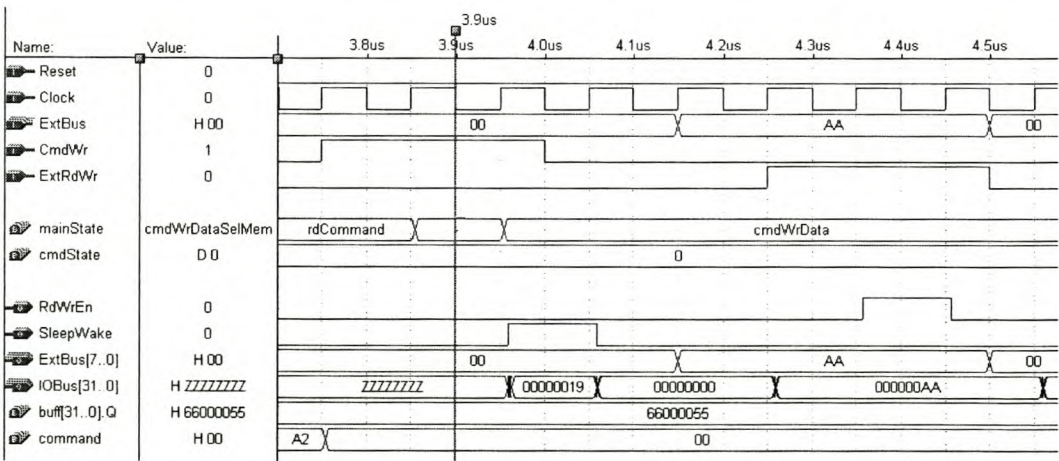


Figure 28 – Router Write Cycle

5.3.2 GMI Simulation

5.3.2.1 GMI Address Set-up

Figure 29 is the simulation for the address set-up cycle. When 'SleepWake' is asserted the GMI's state machine resets. On the next rising clock edge the ID is read and the command (02) is set.

On the next rising clock edge the state changes to 'cmdSetAddr' and the address (0000AAFFH) is read from the IO bus. This address is stored in an internal buffer, which is connected to the 'AddrOut' bus. This is evident from 0000AAFFH appearing on the bus after the clock edge.

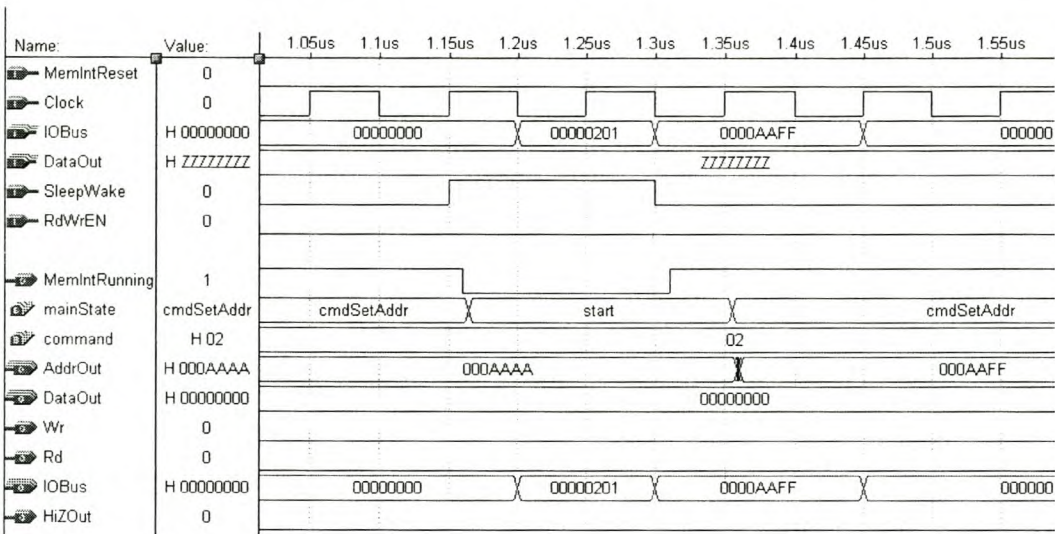


Figure 29 – GMI Address Set-up Simulation

5.3.2.2 GMI Write and Read Cycle

Figure 30 shows how the GMI is selected first with 'command' = 'CwrData' (00) and then 'CrdData' (01). In 'CwrData' mode the value on the IO bus appear on the 'DataOut' bus. Note that if 'Wr' is high, a positive clock edge signals a write to the MSI. After each rising edge the address is incremented by one. 55H and 66H was written to 000AAFFH and 000AB00H respectively.

With the write cycle the IO bus is tri-stated, and when 'Rd' is high the value at 000AB01H will be clocked out and put onto the IO bus to be read by the data router at the next rising clock edge. For this simulation 44H was written to the 'DataOut' bus, and this appeared on the IO bus.

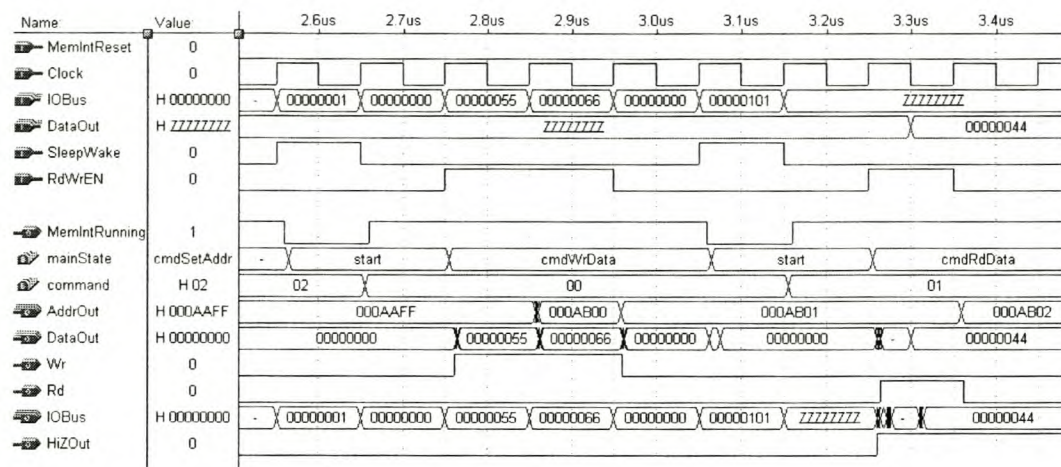


Figure 30 – GMI Write and Read Cycle Simulation

5.3.3 Data Router with a GMI Connected Simulation

Figure 31 is the full simulation including an address set-up, a write, a read and a power control cycle. Each cycle will be discussed with the relevant part of the simulation enlarged. Please refer to Figure 31 to see where each part fits.

The following are system wide simulations using Figure 17 as the top layer design file. The data router, power control and parallel to serial converter are all designated to one FPGA. The memory interface (GMI) and serial to parallel converter are both assigned to their own FPGA's. This is the configuration in which the system will be implemented in hardware. The serial to parallel converter must have a faster device to deal with the 130 MHz serial clock. The other systems do not need fast devices, and cheaper ones can be used.

The previous simulations were performed on each component separately, and the discussions concentrated on the detail of each cycle. The system wide simulation serves to prove that the memory drive performs correctly.

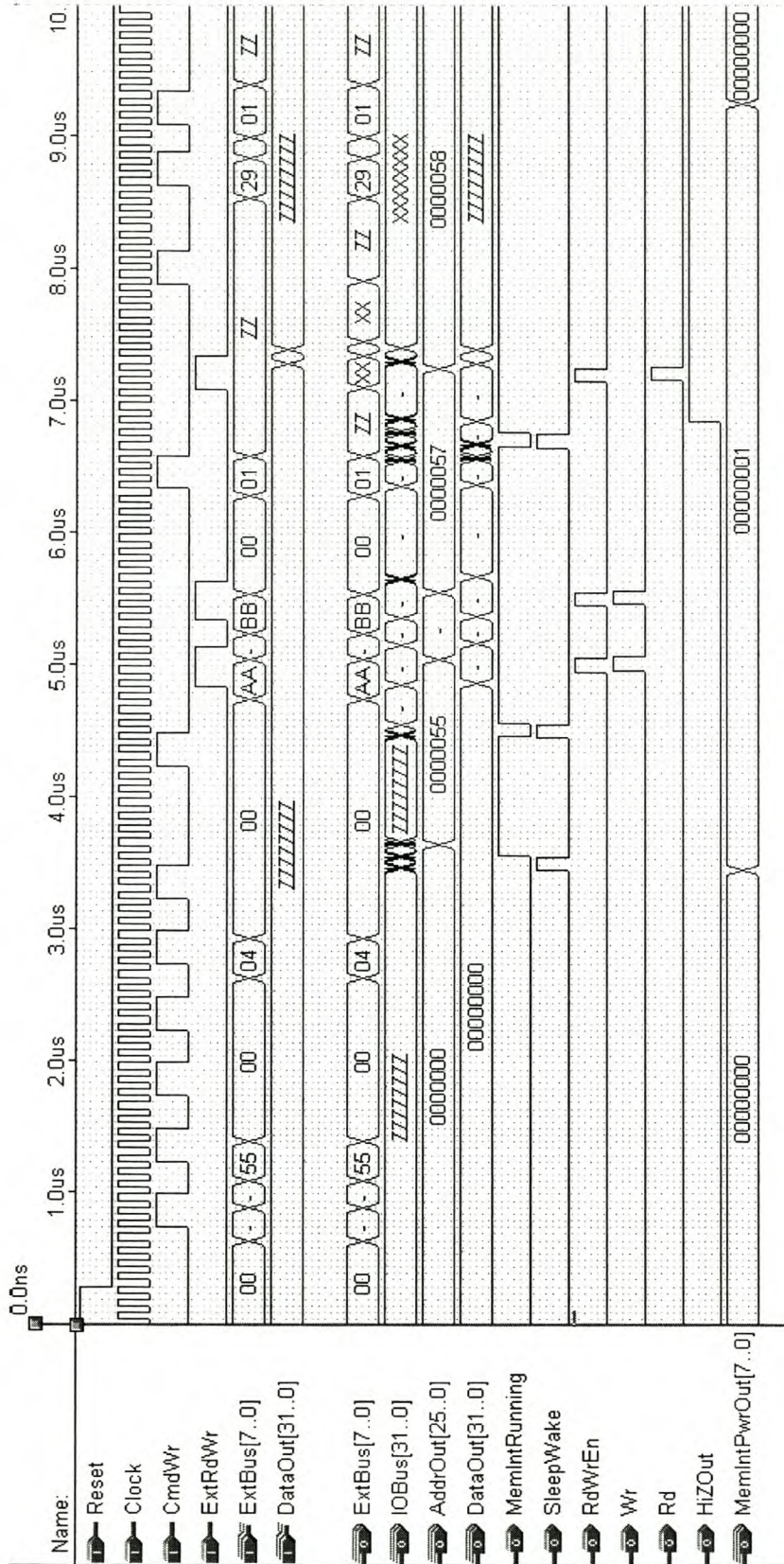


Figure 31 – Data Router with a GMI Connected Simulation

5.3.3.1 System Wide Address Set-up Simulation

Figure 32 shows that the address set-up works as expected. At 3.515 us 201H is written to the IO bus, and then the offset address for the GMI. 'MemIntRunning' indicates that the GMI is active, and this is confirmed with 0000055H appearing on the 'AddrOut' bus. 'mIntPowerOut' will drive the power switches in an implementation of the memory drive. The address cycle automatically switches the GMI on.

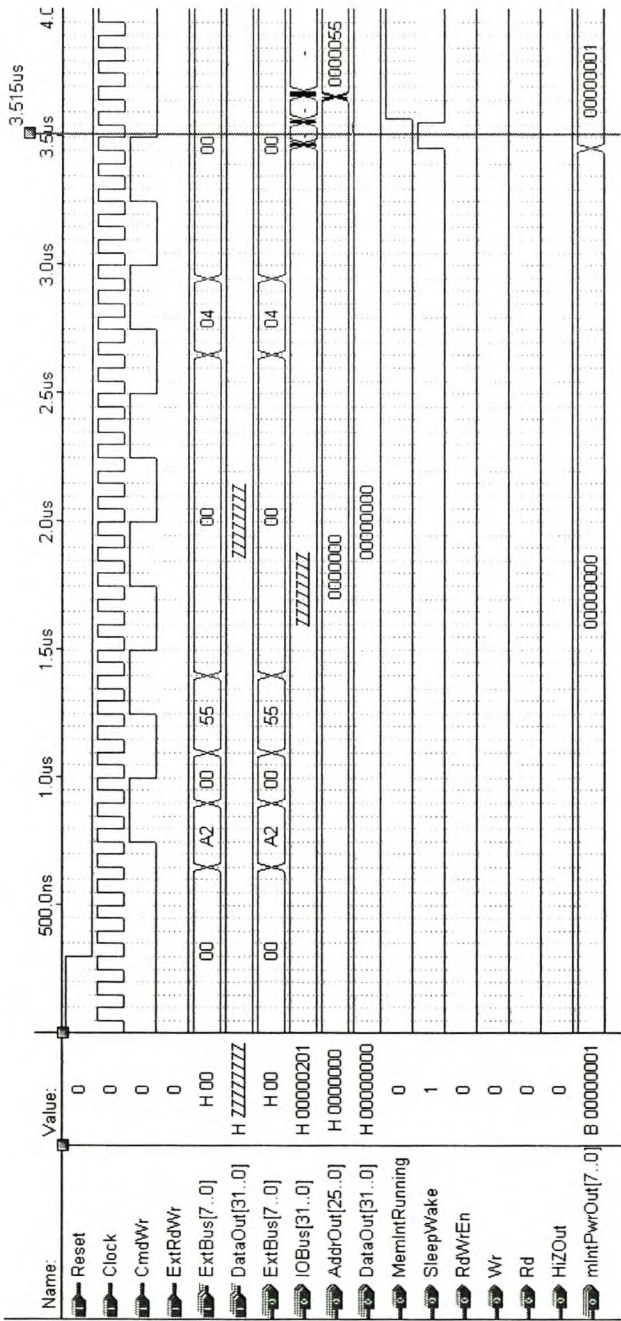


Figure 32 – System Wide Address Set-up Simulation

5.3.3.2 System Wide Write Cycle Simulation

In Figure 33 at 4.52 us the GMI is selected with a write command. Two writes occur correctly, and the address increments. 'Wr' asserts when the correct data is on the 'DataOut' bus.

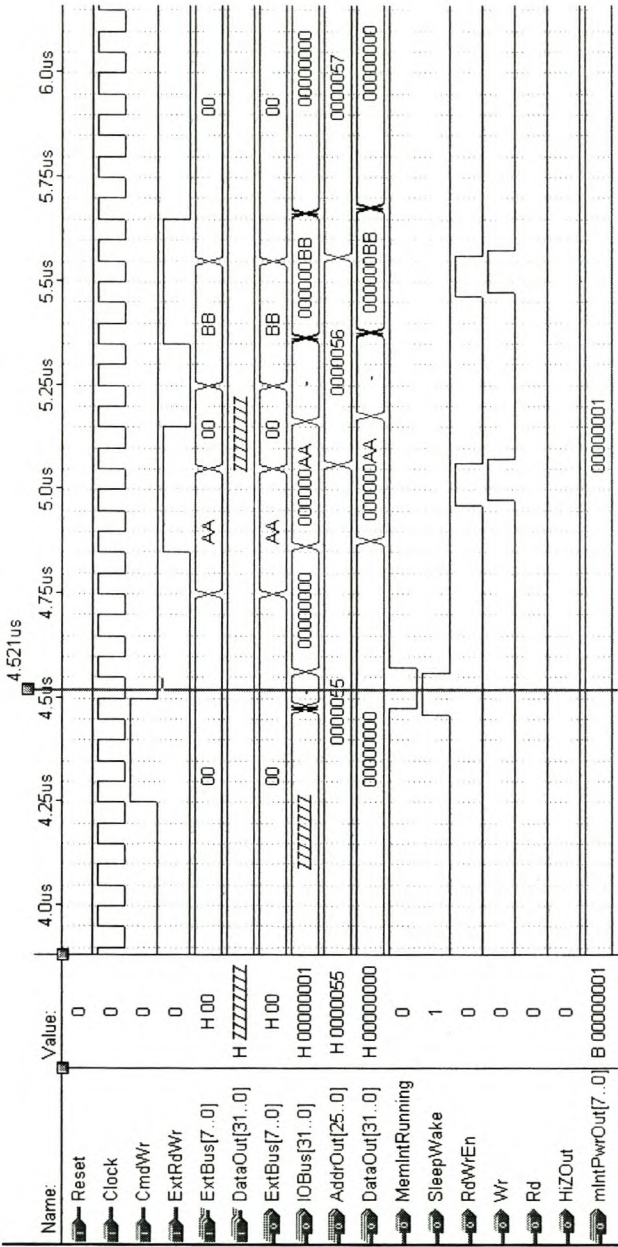


Figure 33 – System Wide Write Cycle Simulation

5.3.3.3 System Wide Read Cycle Simulation

In Figure 34 the GMI is selected with a read command. As the GMI becomes active, 'MemIntRunning' goes high, the data router tri-states the IO bus. Figure 35 shows how 'Rd' is asserted after an 'ExtRdWr' event. At 7.35 us the byte of data that appeared on the 'DataOut' bus is routed to 'ExtBus', where the OBC can then read the byte.

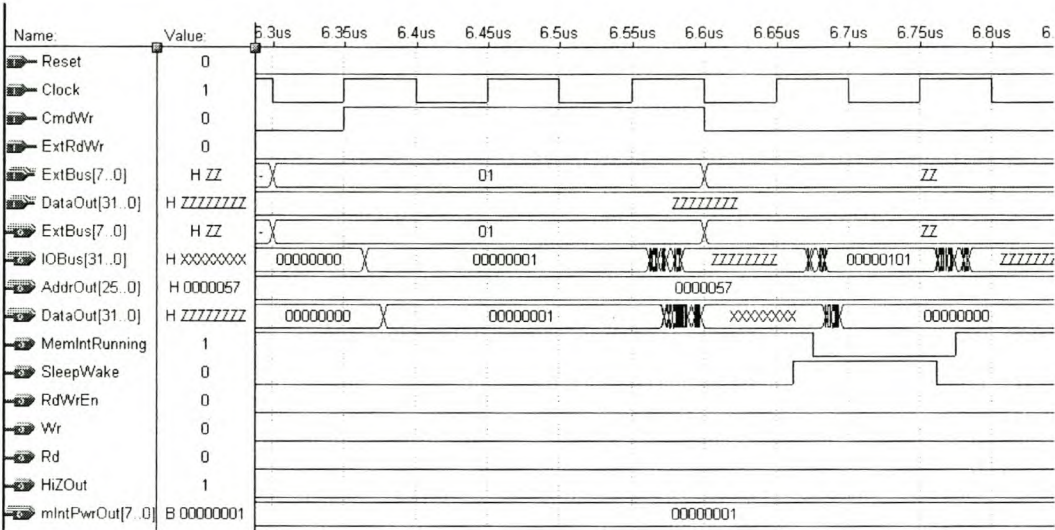


Figure 34 – System Wide Read Cycle Select GMI Simulation

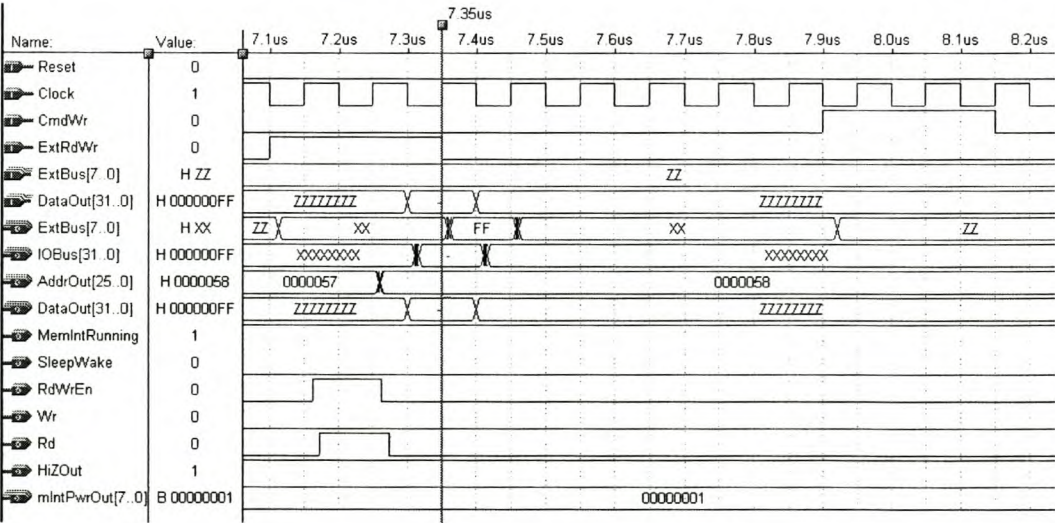


Figure 35 – System Wide Read Simulation

5.3.3.4 Power Control Simulation

In Figure 36 the command 'CforceOff' (09) is written by the OBC. The command register state machine now waits for another 'CmdWr' event. This indicates that the first GMI must be switched off. The information is passed to the data router which switches the correct line on 'mIntPwrOut' off. Note that although it is not evident from the simulation, if any other GMI was switched on it would not have been switched off.

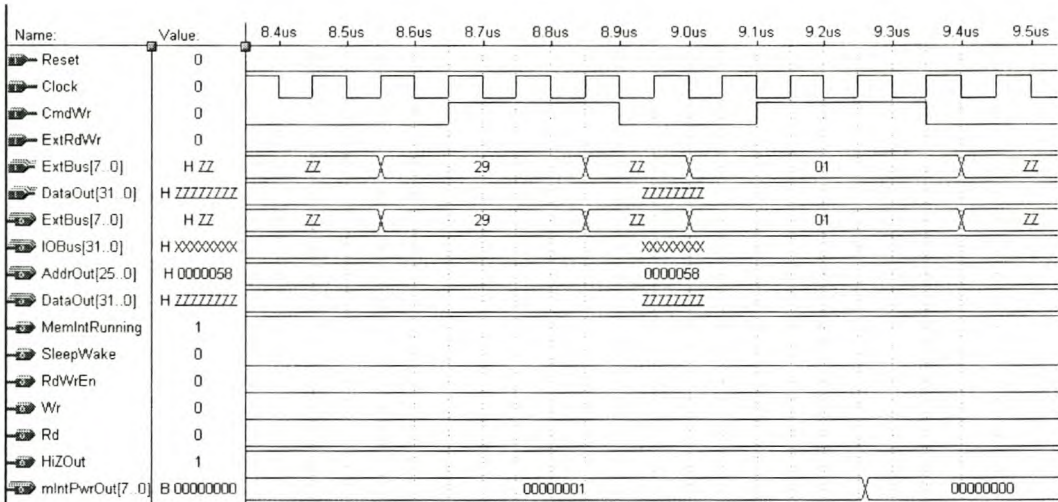


Figure 36 – Power Control Simulation

5.3.3.5 Serial to Parallel Converter Simulation.

Figure 37 illustrates that the serial to parallel converter can operate at 130 MHz. For this simulation the grid size was set to 3.8 ns, which gives a clock frequency of 131 MHz. Every 32 clock pulses 'LatchData' goes high, and the word is latch into the serial buffer by a 'ClkOut0' rising edge. 'ClkOut0' is one of the four clocks which was derived from 'SerDataClk'.

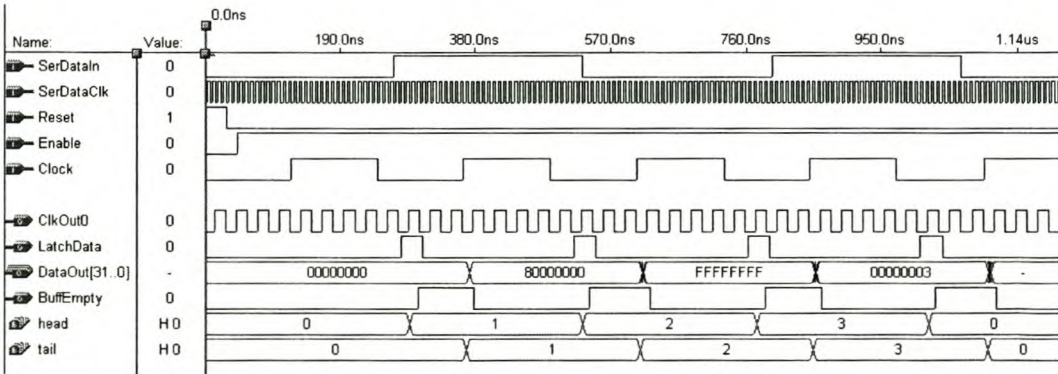


Figure 37 – Serial to Parallel Converter Simulation

At each 'Clock' rising edge the word is read from the buffer and placed on the 'DataOut' bus. The 'DataOut' bus will be connected to the central data router, and the word will be routed to memory from there. The head and tail of the circular buffer indicates when a word is written to the buffer, en when it is read from the buffer.

5.3.3.6 System Wide Serial To Parallel Converter Simulation

In Figure 38 the GMI is reselected with a 'CcapData' (06H) command. This can be seen at 4.5 us. Two clock cycles pass before the serial data reach 'DataOut' (to MSI). These cycles are used to enable the serial to parallel converter. The address increments with four after every write. This is because now 32 bits of data are being written to memory, not 8 bits as in the previous simulation involving OBC data transfers.

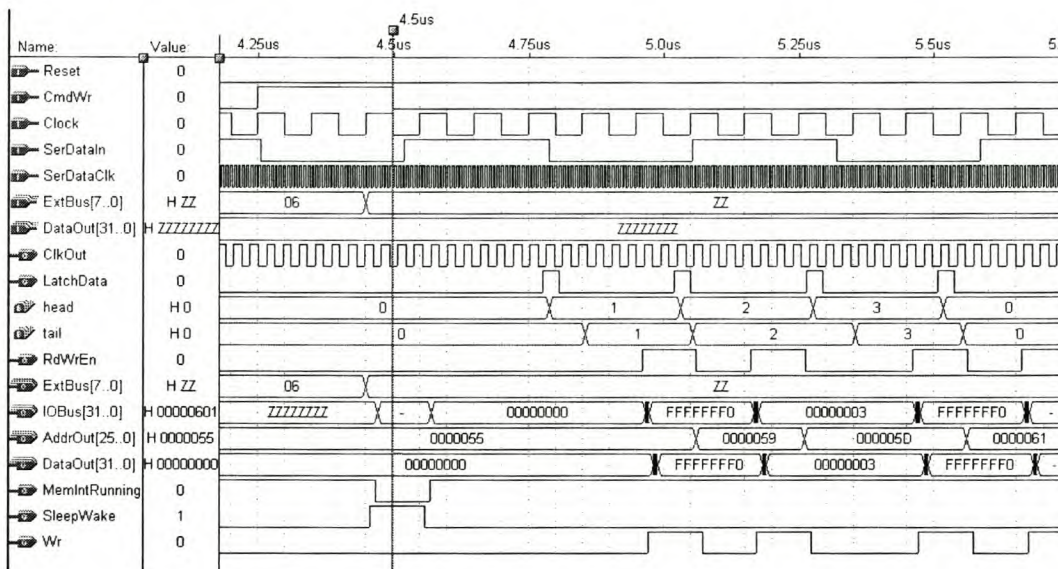


Figure 38 – System Wide Serial To Parallel Converter Simulation

5.3.3.7 System Wide Parallel to Serial Converter Simulation

In Figure 39 at 4.5 us the GMI is reselected with a 'CdumpData' (07H) command. The IO Bus is now tri-stated by the data router, and one clock cycle later the GMI connects the 'DataOut' bus to the IO bus. If 'Rd' is asserted the current memory location's contents will be written to 'DataOut', and the IO bus, by the MSI at the rising clock edge

The parallel to serial converter is enabled, and a 'Load' event occurs every 32 'SerOutClk' cycles. This causes 'RdWrEN' and ultimately 'Rd' to be asserted. When

'RdWrEN' is high and a clock event occurs the data on the IO bus is latched to the parallel to serial converter, where it is loaded into the shift register at the next load event.

The serial clock is running at 50 MHz. It is evident from the simulation that this transfer rate is easily maintained.

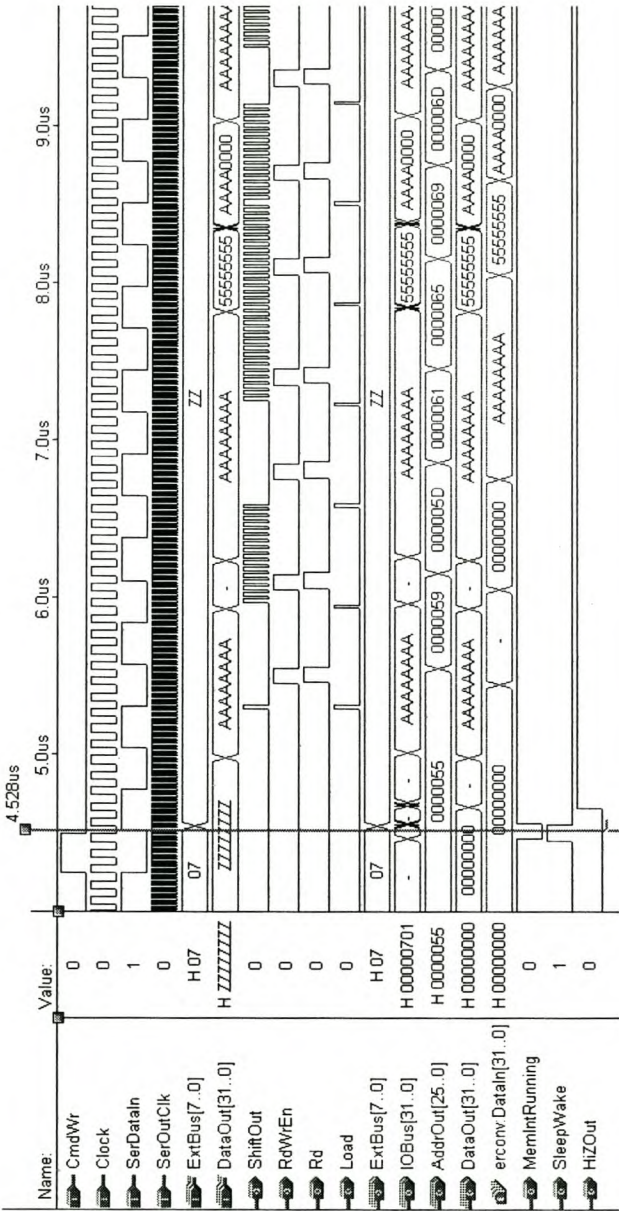


Figure 39 – System Wide Parallel to Serial Conversion Simulation

Chapter 6

CONCLUSION

6.1 Sunsat RAMDISK

The Sunsat RAMDISK was successfully modified to rectify, or bypass all problems encountered during the testing phase. Subsequent flight qualification was also passed. This included exhaustive memory tests, and image capturing.

Sunsat was launched into space on 23 February 1999. The S-Band downlink system failed, and the high-speed downlink functionality of the RAMDISK could never be tested in space. Images were captured though, and sent to earth using alternative paths. These images were deemed a success. Thus the RAMDISK has performed successfully in space, excluding the high-speed downlink, although the fault most probably did not occur on the RAMDISK, but further down the line with the communication hardware.

6.2 Memory Drive

Successful simulations to verify the operation of the design were done. These indicated that the memory drive could be implemented in Altera FPGA's and that a sequential write speed of 130 Mb/s could be maintained.

This project went so far as to design up to the generic memory interface. A memory specific interface was conceptualised, but not designed, nor was physical memory selected. These aspects are covered in [3]. In this thesis he explored memory technologies and also implemented a 64 MB Flash memory module. The control logic was also developed for implementation in a FPGA. This memory module can be used in an implementation of the memory drive. The control logic will act a MSI (memory specific interface). 64 MB Flash combined with the GMI and MSI will make up a memory block. Sixteen of these blocks can then be added to one data router to make up a 1 GB memory drive.

In a system that uses a nine band imager three memory drives can be used to capture the data. Three bands per memory drive. This will bypass the speed problem and add redundancy as well.

6.2.1 Implementation Notes and Suggestions

6.2.1.1 Different implementations settings

The constants that have to be changed if a different implementation is made are the following:

- CmemIntWidth – The new address width of the GMI's must be changed in the command register, as well as each GMI.
- CmemIntID – Each GMI's ID must be set. This is only changed in each GMI.
- MemIntPwrIn – This vector originates from a decoder. The number of decoder outputs that are monitored must equal the number of GMI's in the system. This is only done in the power control sub-system.
- MemIntPwrOut – This is the power output vector, and the same changes must be made as above.

6.2.1.2 Redundancy

When implementing a memory drive targeted for a spacecraft redundancy is an important aspect. One option is to have three different IO buses. Each IO bus will have its own set of GMI's. With this structure two modes can be selected. One where data is just stored sequentially and the three structures filled one set of GMI's after the other.

Another mode, which will be reserved for high data reliability, will work as follows: three copies of the same data is written simultaneously to each set of GMI's. When retrieving the data a voting algorithm is employed where the three sets are compared to each other and if two are the same the win the vote and is taken as the data [7]. This implies that three times the amount of memory is needed to store data. But the full capacity can still be used for less reliable data storage, like an image.

6.2.2 Further Research

The most important next step is to load the design into silicon and verify real world operation. This implies that a board will have to be designed that not only contains the required FPGA's, but also the peripheral devices needed to support the core memory drive.

Different programmable logic devices must also be investigated, especially for the harsh space environment. Here the fuse-link option from ACTEL could be a solution.

A new study must be done to determine the best memory type to use when a memory drive is implemented. A memory specific interface will also have to be designed to interface with the chosen memory. A company called "3D-Plus Electronics" have come up with a memory manufacturing technique which is highly suited for mass data recorders with space and weight constraints. The memory is stacked in three-dimensional cubes, which implies that large amounts of memory take up relatively little space. They manufacture DRAM, FLASH and SRAM modules. A one Gbit DRAM device measuring $13.8 \times 24.2 \times 7.75$ mm is currently available.

REFERENCES

- [1] Altera, *Data Book*, 1996
- [2] Altera, *The Flex 8000 Handbook*, 1994
- [3] Badenhorst P.J., *The Development of a Memory System for the second generation SUNSAT micro-satellite*, M-Thesis, University of Stellenbosch, 1997
- [4] Burger H.S., *Development of a Memory-Module for The SUNSAT Micro-Satellite*, M-Thesis, University of Stellenbosch, 1996
- [5] Burger H.S., *Schematics of the Memory-Module for the SUNSAT micro-satellite*, SUNSAT work package, 1995
- [6] Chan Pak K., Mourad S., *Digital Design Using Field Programmable Gate Arrays*, Prentice Hall, 1994
- [7] Johnson B. W., *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley, 1989
- [8] Koekemoer J.A.K., *A CAN-Based Command & Data Handling System For Future SUNSAT micro-satellites*, M-Thesis, University of Stellenbosch, 2000
- [9] le Roux A.G., *An FPGA-based Approach to the Compacting of the SUNSAT TTM System*, M-Thesis, University of Stellenbosch, 1998
- [10] Milne G.W., *SUNSAT Technical Specifications*, Internal report, Electrical and Electronic Engineering Department, University of Stellenbosch, September 1991
- [11] Naylor D., Jones S., *VHDL: A Logic Synthesis Approach*, Chapman & Hall, 1997
- [12] Oldfield J., Dorf R., *Field Programmable Gate Arrays*, John Wiley & Sons Inc., 1995
- [13] SYNOPSYS, *VHDL compiler Reference Manual V3.5*, 1996
- [14] Wakerly J.F., *Digital Design: Principles and Practices*, Prentice-Hall, Inc., 1990
- [15] Wertz J.R., Larson W.J., *Space Mission Analysis and Design*, Microcosm Press, Kluwer Academic Publishers, 1999

Appendix A

MEMORY DRIVE SIGNAL LISTINGS

A.1 Introduction

Here the signals used in the memory drive sub-systems will be listed. A description of each will also be included. Refer to these descriptions when studying the simulations in Chapter 5 and VHDL code in Appendix B.

The naming convention used: Input/output signal start with upper case letters; internal signals start with lower case letters. Constant signals start with an upper case 'C'.

A.2 Signal Listings

A.2.1 Data Router and Command Register Signals

A.2.1.1 Input/Output Signals

A.2.1.1.1 General I/O

Signal	Width/Type	Description
Clock	Bit	This is the system clock that drives the state machines. To meet the speed requirement of 130 Mbits/s the minimum frequency for this clock is 5 MHz.
Reset	Bit	Global Reset. If one of the state machines are stuck in a state due to incorrect command write procedures this signal will reset everything.

A.2.1.1.2 Command Register I/O

Signal	Width/Type	Description
ExtBus	8 Bits	This is a bi-directional interface to the OBC, or telcommand communications processor.
CmdWr	Bit	By pulsing this signal commands on the 'ExtBus' are latched into the command register.
ExtRdWr	Bit	A rising flank initiates a data read or write cycle. This data is read from memory/written to memory to/from the 'ExtBus'.

A.2.1.1.3 Generic memory interface I/O

Signal	Width/Type	Description
IOBus	32 Bits	This is the general I/O data bus between the data router and all the GMI's. Address and Data are both carried on this bus. Both the data router and memory interfaces can write data to this bus.
RdWrEn	Bit	If this signal is high data is read/written from/to memory on the rising edge of the system clock.
SleepWake	Bit	'SleepWake' is used to select GMI's. If high all the GMI's reads the least significant byte from the 'IOBus' on a rising system clock edge. If the byte matches a GMI's ID that GMI becomes active. When this signal goes high the active GMI's state machine reset and if it is not selected again it remains inactive.

A.2.1.1.4 Serial to Parallel Converter I/O

Signal	Width/Type	Description
DataIn	32 Bits	Data from the circular buffer is placed on this bus for the data router to read.
BuffEmpty	Bit	This signal indicates when the buffer has on or more elements available. If it is high the head and tail pointer of the circular buffer do not point to the same location, and data is ready for collection.
EnableSerTPar	Bit	The data router only enables the serial to parallel converter if a capture command is set. This signal enables and disables serial to parallel converter.

A.2.1.1.5 Parallel to Serial Converter I/O

Signal	Width/Type	Description
DataOut	32 Bit	When a dump data command is set data is routed from the 'IOBus' to 'DataOut' which is connected to the parallel to serial converter.
Load	Bit	A rising 'Load' edge indicates new data is needed on 'DataOut' to serialise.
EnableParTSer	Bit	The subsystem is only enabled if a data dump is being performed.

A.2.1.1.6 Power Control I/O

Signal	Width/Type	Description
--------	------------	-------------

MemIntPwr	9 Bits	This indicates which memory interface must be switched on or off. The 9 th bit indicates whether to switch it on or off. One is added to the GMI ID, and this nr. is assigned to 'MemIntPwr', which is connected to a decoder. The decoder's LSB is ignored, because the default state for 'MemIntPwr' is 0, and this does not indicate that the 0 th GMI must be switch on. This is why one is added, and the decoder output is shifted by one. The implication is that there can only be 255 GMI's.
SwitchPwr	Bit	This signal is asserted when the indicated GMI must be switched on or off.

A.2.1.2 Internal Signals

A.2.1.2.1 Signals used to interface with the Parallel To Serial Converter

Signal	Width/Type	Description
dumpDataRd	Bit	This is the synchronised version of 'Load', and indicates when a word must be read from memory to be sent to the parallel to serial converter for serialisation.
loadRdy	Bit	The 'Load' monitor process sends this signal to the 'dumpDataRd' process for synchronisation.
enableParTSerI	Bit	The internal enable signal (an output cannot be read, thus an internal signal must be used)
dataOutI	32 Bits	Internal bus connected to 'DataOut'

A.2.1.2.2 Data Router State Types

mainStateTypes	Description
start	The start-up state. All signals are reset to their default values in this state. A reset event forces this state.
rdCommand	This state waits for a command from the communications processor. Here is a case statement that branches to the correct state depending on the command.
cmdSetAddrSelMem	Select the indicated GMI with an address set-up command.
cmdSetAddr	Write the offset address inside the GMI to the GMI.
cmdWrDataSelMem	Select the indicated GMI with a write command.
cmdWrData	Write 8 bit data from the 'ExtBus' to memory.
cmdRdDataSelMem	Select the indicated GMI with a read command.
cmdRdDataRelBus	Release the 'IOBus' for one clock cycle.
cmdRdData	Read 8 bit data to the 'ExtBus' from memory.
cmdCapDataSelMem	Select the indicated GMI with a capture data command.
cmdCapData	Write 32 bit data from the 'DataIn' bus to memory.
cmdDumpDataSelMem	Select the indicated GMI with a dump data command.

cmdDumpDataRelBus	Release the 'IOBus' for one clock cycle.
cmdDumpData	Read 32 bit data to the 'DataOut' bus from memory.

A.2.1.2.3 Data Router Internal Signals

Signal	Width/Type	Description
CiOBUSDontCare	32 bit Constant	Because the memory interface size can be changed this vector is used with a constant index to always assign the correct amount of bits to a bus.
mainState	mainState Types	The signal that controls the state machine.
iOBUSI	32 Bits	Internal signal for the 'IOBus'.
memInt	8 Bits	The signal used to store the current GMI ID.
memIntPwrI	9 Bits	Internal GMI on/off vector.
capCounter	38 Bits	This counter is set-up with the amount of 32 bit words to capture during a data capture operation. With each word captured it is decremented, and stop the capture operation when it reaches zero. It is 38 bits wide to be able to address one Terabyte of 32 bit words.
hiZ	Bit	Controls the tri-state buffer for the 'IOBus'
cmdAck	Bit	Acknowledges that the command has been executed.
extRdWrAck	Bit	Acknowledges that the read/write operation has been performed.
stopCap	Bit	Used in conjunction with 'capCounter' to stop the capture operation when the counter reaches zero.

A.2.1.2.4 Command Register Static Variable and Command Table

Signal	Width/Type	Description
CmemIntWidth	26 Bit Constant	This constant defines the memory interface size. For this simulation it was wet to 26 bits => 64 Meg / memory Interface Max = 32 bits because of bus width => 4 Gig
CwrData	5 Bits	This constant is described in Table 2 – Command Code Table.
CrdData	5 Bits	"
CsetAddr	5 Bits	"
CgetAddr	5 Bits	"
CwrCmd	5 Bits	"
CrdStatus	5 Bits	"
CcapData	5 Bits	"
CdumpData	5 Bits	"
CforceOn	5 Bits	"
CforceOff	5 Bits	"

A.2.1.2.5 Command Register State Types

cmdStateTypes	Description
rdCmd	This is the start-up, and default state. In this state the command register waits for a new command.
rdBufferOne	This first byte to follow the command is read.
rdBufferTwo	This second byte to follow the command is read.
rdBufferThree	This third byte to follow the command is read.
rdBufferFour	This fourth byte to follow the command is read.
rdBufferFive	This fifth byte to follow the command is read.

A.2.1.2.6 Command Register Internal Signals

Signal	Width/Type	Description
cmdState	cmdState Types	Controls the command register state machine
nextCmdState	cmdState Types	This state machine is implemented differently from the rest, and need a variable to hold the next state.
command	8 Bits	The command is stored in this buffer
commandI	8 Bits	Interim storage for a command until clock edge loads it into 'command'
bufferOne	8 Bits	Interim storage for the first byte to follow the command until a clock edge loads it into the correct position in 'buff'.
bufferTwo	8 Bits	Interim storage for the second byte to follow the command until a clock edge loads it into the correct position in 'buff'.
bufferThree	8 Bits	Interim storage for the third byte to follow the command until a clock edge loads it into the correct position in 'buff'.
bufferFour	8 Bits	Interim storage for the fourth byte to follow the command until a clock edge loads it into the correct position in 'buff'.
bufferFive	8 Bits	Interim storage for the fifth byte to follow the command until a clock edge loads it into the correct position in 'buff'.
buff	40 Bits	Storage for the five bytes (parameters for the received command).
cmdRdy	Bit	Indicates to the data router state machine that a command is ready to be executed.
cmdRdyI	Bit	Internal command ready signal to be latched into 'cmdRdy' at a clock edge.
extRdWrRdy	Bit	Indicates to the data router state machine that a byte is ready to be read/written from/to 'ExtBus'.
extHiZ	Bit	Controls the tri-state buffer for 'ExtBus'.

extHiZForce	Bit	Forces the 'ExtBus' tri-state buffer into tri-state.
extBusI	8 Bits	Internal bus for 'ExtBus'.

A.2.2 Generic Memory Interface Signals

A.2.2.1 Input/Output Signals

A.2.2.1.1 General I/O

Signal	Width/Type	Description
Clock	Bit	System clock that drives the state machine.
MemIntReset	Bit	This resets the GMI, but does not modify the address. Thus if a state machine is stuck in a state due to incorrect command format the system can be reset, and operation can resume without re-setting the address.
OverFlowIn	Bit	Indicates that an overflow from a preceding GMI has occurred.
OverFRdWrIn	Bit	Indicates which mode the previous GMI was in: Read operation=1; Write operation = 0.
OverFBOrWIn	Bit	Indicates which mode the previous GMI was in: 32 bit operation=1; 8 bit operation = 0.
OverFlowOut	Bit	Signals the next GMI to activate and continue read/write operations. This is the MSB + 1 of the address.
OverFRdWrOut	Bit	Indicates read/write mode, see above.
OverFBOrWOut	Bit	Indicates byte/word mode, see above.
Alive	Bit	This line is used to select whether to send the overflow to this device, or skip this device. If it is zero, the device will be skipped.

A.2.2.1.2 Central Data Router I/O

Signal	Width/Type	Description
IOBus	32 Bits	See A.2.1.1.3 Generic memory interface I/O.
SleepWake	Bit	See A.2.1.1.3 Generic memory interface I/O.
RdWrEN	Bit	See A.2.1.1.3 Generic memory interface I/O.

A.2.2.1.3 Memory Specific Interface I/O

Signal	Width/Type	Description
DataOut	32 Bits	This is a bi-directional bus that connects the memory specific interface with the GMI. Data that is written/read to/from memory is transferred on this bus. Note that the number of the GMI will be

		appended to the end of this name e.g. the Data bus for GMI nr. 1 is 'DataOut1'.
AddrOut	26 Bits	The address for the next write/read operation. Note that the number of the GMI will be appended to the end of this name e.g. the Address bus for GMI nr. 1 is 'AddrOut1'.
Rd	Bit	If high the MSI must perform a read operation at the next rising clock edge.
Wr	Bit	If high the MSI must perform a write operation at the next rising clock edge.

A.2.2.1.4 Testing Outputs

Signal	Width/Type	Description
HiZOut	Bit	Indicates when the 'IOBus' is driven by the GMI.
MemIntRunning	Bit	Indicates when the GMI is active.

A.2.2.2 Internal Signals

A.2.2.2.1 Constants and Command Table

Signal	Width/Type	Description
CmemIntID	8 Bit Constant	This constant contains the GMI's ID. This ID is obtained from the segment in the address.
CmemIntWidth	26 Bit Constant	The address width in bits of the GMI.
CaddrZeroVect	32 Bit Constant	A zero vector which is used to zero the address. 'CmemIntWidth' is used as an index to assign the correct amount of zeros to the 'addr' vector.
CwrData	5 Bits	This constant is described in Table 2 – Command Code Table.
CrdData	5 Bits	"
CsetAddr	5 Bits	"
CgetAddr	5 Bits	"
CcapData	5 Bits	"
CdumpData	5 Bits	"
CforceOn	5 Bits	"
CforceOff	5 Bits	"

A.2.2.2.2 Memory Interface State Types

mainStateTypes	Description
start	Start is the default state for the GMI state machine. This state contains a case statement which jumps to the correct state depending on the mode read when the GMI was activated.

cmdSetAddr	Writes the value on the 'IOBus' to the address buffer.
cmdWrData	Passes data from the 'IOBus' to the 'DataOut' bus, and connects 'RdWrEN' with 'Wr'. The address is incremented by 1.
cmdRdData	Passes data from the 'DataOut' bus to the 'IOBus', and connects 'RdWrEN' with 'Rd'. The 'IOBus' tri-state buffer is disabled, thus the GMI drives the 'IOBus'. The address is incremented by 1.
cmdCapData	Same as 'cmdWrData', but the address is incremented by 4 because this is a 32 bit operation.
cmdDumpData	Same as 'cmdRdData', but the address is incremented by 4 because this is a 32 bit operation.

A.2.2.2.3 Memory Interface Internal Signals

Signal	Width/Type	Description
mainState	mainState Types	The GMI's state machine control variable.
mainStateI	mainState Types	When an overflow occurs the correct state is immediately assigned to 'mainState', but in normal operations this signal is assigned to 'mainState'.
iOBUSI	32 Bits	Internal 'IOBus'.
dataOutI	32 Bits	Internal 'DataOut' bus.
addr	Bit	GMI's address buffer. This buffer is set with the address during an address set-up operation, and subsequent reads or writes are done at this address in memory. It is incremented automatically after every operation. It is also zeroed when an overflow occurs.
command	5 Bits	Stores the command received when the GMI was selected.
running	Bit	This is the active signal for the GMI. When 'running' = 1 the GMI is active. It is either activated by a 'SleepWake' event, or an overflow event.
runningMntr	Bit	This signals 'running' that the correct ID has been received.
runningOFlw	Bit	This signals 'running' that an overflow has occurred.
hiZ	Bit	Controls the tri-state buffers for the 'IOBus'.
zeroAddrAck	Bit	Acknowledges that the address has been zeroed.
zeroAddr	Bit	Indicates that the address must be zeroed. This occurs when an overflow event has happened.

A.2.3 Power Control Signals

A.2.3.1 Input/Output Signals

A.2.3.1.1 General I/O

Signal	Width/Type	Description
PwrReset	Bit	Reset signal.

A.2.3.1.2 Central Data Router I/O

Signal	Width/Type	Description
MemIntPwrIn	8 Bits	For this simulation 8 GMI's are controlled by the power control subsystem. If more memory interfaces are added the bus width must be updated to reflect the number of GMI's. One bit for each GMI. There can be a maximum of 255 GMI's
SwitchPwr	Bit	A rising flank indicates to switch the power on/off for the indicated GMI.
OnOff	Bit	Indicates on/off switch
MemIntPwrOut	8 Bits	Output vector that drives the power switches for each GMI.

A.2.3.1.3 Internal Signal

Signal	Width/Type	Description
memIntPwrOutI	8 Bits	Internal output GMI power vector.

A.2.4 Parallel to Serial Converter Signals

This subsystem does not contain a VHDL file, and the following signals are from the 'GDF' file, see Figure 25.

Signal	Width/Type	Description
DataIn	32 Bits	The 32 bit word to be serialised is written to this bus by the data router.
SerOutClk	Bit	This clock dictates the speed at which the data is serialised.
Enable	Bit	Enables/disables the parallel to serial converter.
Reset	Bit	Reset signal
ShiftOut	Bit	The serialised data is available on this line.

Load	Bit	When a new word needs to be loaded this signal sends a pulse to the data router.
------	-----	--

A.2.5 Serial to Parallel Converter Signals

This subsystem consists of two modules: the shift register (SHIFTRREG) and circular buffer (CIRCBUFF). See Figure 22.

A.2.5.1 Shift Register Signals

Signal	Width/Type	Description
SerDataClk	Bit	The incoming serial data clock.
SerDataIn	Bit	The incoming serial data.
Enable	Bit	Enables/disables this sub-system.
Reset	Bit	Reset signal
ClkOut0	Bit	This is one of the four slower clocks which is derived from the 'SerDataClk' and is used to store the word in the circular buffer.
LatchData	Bit	Indicates that on a rising edge of 'ClkOut0' the data word must be stored.
DataOut	32 Bits	The shifted data word.

A.2.5.2 Circular Buffer Signals

A.2.5.3 Input/Output Signals

A.2.5.3.1 Genral I/O

Signal	Width/Type	Description
Clock	Bit	System Clock
SerDataClk	Bit	This clock is used to store the incoming data word.
Enable	Bit	Enables/disables the circular buffer.
Reset	Bit	Reset signal.

A.2.5.3.2 Central Data Router I/O

Signal	Width/Type	Description
DataOut	32 Bits	This bus carries the data which is read from the circular buffer.
BuffEmpty	Bit	Indicates when there is data in the circular buffer. This signal is derived from the head an tail

		counters.
--	--	-----------

A.2.5.3.3 Shift Register I/O

Signal	Width/Type	Description
DataIn	32 Bits	Incoming data word.
LatchData	Bit	Indicates that on a rising edge of 'SerDataClk' the data word must be stored.

A.2.5.4 Internal Signals

Signal	Width/Type	Description
BuffElement	32 Bits	A subtype declaration of a 32 bit vector.
BuffArray	Array	An array of 32 bit vectors. This array contains 4 elements.
dataOutI	32 Bits	Internal 'DataOut' buffer.
head	Bit	Head counter. The incoming data is stored at this location in the 'BuffArray'.
tail	Bit	Tail counter. The outgoing data is read from this location in the 'BuffArray'.
buff	BuffArray	The circular buffer storage array.

A p p e n d i x B

MEMORY DRIVE VHDL CODE

B.1 Data Router And Command Register (Router.vhd)

```
-- Memory Drive Data Router and Command Register

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY router IS

    PORT (
        -- General I/O
        Clock      : IN STD_LOGIC;
        Reset      : IN STD_LOGIC;
        -- Command Register I/O
        ExtBus      : INOUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        CmdWr       : IN STD_LOGIC;
        ExtRdWr     : IN STD_LOGIC;
        -- Generic memory interface I/O
        IOBus       : INOUT STD_LOGIC_VECTOR(31 DOWNTO 0);
        RdWrEn      : OUT STD_LOGIC;
        SleepWake   : OUT STD_LOGIC;
        -- Serial to Parallel Converter I/O
        DataIn      : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        BuffEmpty   : IN STD_LOGIC;
        EnableSerTPar : OUT STD_LOGIC;
        -- Parallel to Serial Converter I/O
        DataOut     : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
        Load        : IN STD_LOGIC;
        EnableParTser : OUT STD_LOGIC;
        -- Power Control I/O
        MemIntPwr   : OUT STD_LOGIC_VECTOR(8 DOWNTO 0);
        SwitchPwr   : OUT STD_LOGIC
    );
END router;

ARCHITECTURE a OF router IS
    -- Parallel To Serial Converter
    SIGNAL dumpDataRd : STD_LOGIC;
    SIGNAL loadRdy    : STD_LOGIC;
    SIGNAL enableParTserl : STD_LOGIC;
    SIGNAL dataOutl   : STD_LOGIC_VECTOR(31 DOWNTO 0);

    -- Data Router
    CONSTANT CiOBUSdontCare : STD_LOGIC_VECTOR(31 DOWNTO 0) :=
"XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";

    TYPE mainStateTypes IS (start, rdCommand, cmdSetAddrSelMem, cmdSetAddr, cmdWrDataSelMem,
        cmdWrData, cmdRdDataSelMem, cmdRdDataRelBus, cmdRdData, cmdCapDataSelMem,
        cmdCapData, cmdDumpDataSelMem, cmdDumpDataRelBus, cmdDumpData);

    SIGNAL mainState : mainStateTypes;

    SIGNAL iOBUSl : STD_LOGIC_VECTOR(31 DOWNTO 0);
    SIGNAL memInt : STD_LOGIC_VECTOR(7 DOWNTO 0);
    SIGNAL memIntPwrl : STD_LOGIC_VECTOR(8 DOWNTO 0);
    SIGNAL capCounter : STD_LOGIC_VECTOR(37 DOWNTO 0);

    SIGNAL hiZ : STD_LOGIC;
    SIGNAL cmdAck : STD_LOGIC;
    SIGNAL extRdWrAck : STD_LOGIC;
    SIGNAL stopCap : STD_LOGIC;

    -- Command Register
```

```
-- Static Variables

-- Define memory memInt size = 26 bits => 64 Meg / memory Interface
-- Max = 32 bits because of bus width => 4 Gig
-- If more needed add extra state
CONSTANT CmemIntWidth : INTEGER := 26; -- Nr of bits in each memInt

-- Command Table
CONSTANT CwrData : STD_LOGIC_VECTOR(4 DOWNTO 0) := "00000"; -- Wr Data to Ram
CONSTANT CrdData : STD_LOGIC_VECTOR(4 DOWNTO 0) := "00001"; -- Rd Data from Ram
CONSTANT CsetAddr : STD_LOGIC_VECTOR(4 DOWNTO 0) := "00010"; -- Setup Address
CONSTANT CwrCmd : STD_LOGIC_VECTOR(4 DOWNTO 0) := "00100"; -- Wr Cmd to memInt
CONSTANT CrdStatus : STD_LOGIC_VECTOR(4 DOWNTO 0) := "00101"; -- Rd Status from memInt
CONSTANT CcapData : STD_LOGIC_VECTOR(4 DOWNTO 0) := "00110"; -- Capture Image Data
CONSTANT CdumpData : STD_LOGIC_VECTOR(4 DOWNTO 0) := "00111"; -- Dump Block of Data
CONSTANT CforceOn : STD_LOGIC_VECTOR(4 DOWNTO 0) := "01000"; -- Force memInts On
CONSTANT CforceOff : STD_LOGIC_VECTOR(4 DOWNTO 0) := "01001"; -- Force memInts Off

TYPE cmdStateTypes IS (rdCmd, rdBufferOne, rdBufferTwo, rdBufferThree, rdBufferFour, rdBufferFive);

SIGNAL cmdState : cmdStateTypes;
SIGNAL nextCmdState : cmdStateTypes;

SIGNAL command : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL commandI : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL bufferOne : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL bufferTwo : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL bufferThree : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL bufferFour : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL bufferFive : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL buff : STD_LOGIC_VECTOR(39 DOWNTO 0);

SIGNAL cmdRdyI : STD_LOGIC;
SIGNAL cmdRdy : STD_LOGIC;
SIGNAL extRdWrRdy : STD_LOGIC;
SIGNAL extHiZ : STD_LOGIC;
SIGNAL extHiZForce : STD_LOGIC;
SIGNAL extBusI : STD_LOGIC_VECTOR(7 DOWNTO 0);

BEGIN
-- Parallel To Serial Converter

-- This process signals the router that the parallel to serial converter has loaded the 32 bit data
-- word and is waiting for new data.
LoadRdySignal:
PROCESS (Load, Reset, dumpDataRd)
BEGIN
    IF Reset = '1' OR dumpDataRd = '1' THEN -- If dumpDataRd = '1' then the data word has been sent.
        loadRdy <= '0';
    ELSIF Load'EVENT AND Load = '1' THEN
        loadRdy <= '1';
    END IF;
END PROCESS LoadRdySignal;

-- Synchronise the load event with the clock driving the router state machine, and generate a clean pulse to
-- read data from memory
SyncLoadEvent:
PROCESS (Clock, Reset)
BEGIN
    IF Reset = '1' OR enableParTserI = '0' THEN -- enableParTserI inhibits this process if it is not needed
        dumpDataRd <= '0';
    ELSIF Clock'EVENT AND Clock = '1' THEN -- Synchronise Load event with clock
        IF loadRdy = '1' THEN
            dumpDataRd <= '1';
        ELSE
            dumpDataRd <= '0';
        END IF;
    END IF;
END PROCESS SyncLoadEvent;

-- Buffered bus connected to the Output bus
DataOut <= dataOutI;
```



```

-- EnableParTSerI needs to be read, thus a signal must be used
EnableParTSerI <= EnableParTSerI;

-- Router

-- Amount of 32 bit words to capture from Imager
CaptureCounter:
PROCESS (mainState, BuffEmpty)
BEGIN
  IF mainState = cmdCapDataSelMem THEN
    stopCap <= '0';
    capCounter <= buff(37 DOWNT0 0);
  ELSIF BuffEmpty'EVENT AND BuffEmpty = '1' THEN
    IF capCounter = "000000000000000000000000000000000000" THEN
      stopCap <= '1';
      capCounter <= capCounter;
    ELSE
      stopCap <= '1';
      capCounter <= capCounter - '1';
    END IF;
  END IF;
END PROCESS CaptureCounter;

-- The signals are defined for all states so that no unnecessary memory is created to remember a signals previous
state.
MainStateMachine:
PROCESS (Clock, Reset)
BEGIN
  IF Reset = '1' THEN
    mainState <= start;
  ELSIF Clock'EVENT AND Clock = '1' THEN
    CASE mainState IS
      WHEN start =>
        dataOutI <= CiOBUSDontCare;
        iOBUSI <= CiOBUSDontCare;
        extBUSI <= "XXXXXXXX";
        memIntPwri <= "0000000000";
        memInt <= memInt;
        hiZ <= '0';
        sleepWake <= '0';
        rdWrEn <= '0';
        cmdAck <= '0';
        extRdWrAck <= '0';
        EnableSerTPar <= '0';
        enableParTSerI <= '0';
        mainState <= rdCommand;
      WHEN rdCommand =>
        -- Waits until a new command is in the command buffer and then branches to the correct state,
        -- depending on the command.
        IF cmdRdy = '1' THEN
          CASE command(4 DOWNT0 0) IS
            WHEN CwrData =>
              memIntPwri <= memIntPwri;
              SwitchPwr <= '0';
              memInt <= memInt;
              cmdAck <= '0';
              mainState <= cmdWrDataSelMem;
            WHEN CrdData =>
              memIntPwri <= memIntPwri;
              SwitchPwr <= '0';
              memInt <= memInt;
              cmdAck <= '0';
              mainState <= cmdRdDataSelMem;
            WHEN CsetAddr =>
              memIntPwri(8) <= '1'; -- Indicates that the indicated GMI must be switched on
                                  -- The 8 bits above CmemIntWidth are used to determine
                                  -- the Memory Interface's ID
              memIntPwri(7 DOWNT0 0) <= buff((CmemIntWidth + 8) DOWNT0 CmemIntWidth);
              memInt <= buff((CmemIntWidth + 8) DOWNT0 CmemIntWidth);
              SwitchPwr <= '1'; -- Signal power switch on/off
              cmdAck <= '0';
              mainState <= cmdSetAddrSelMem;
            WHEN CcapData =>
              memIntPwri <= memIntPwri;

```

```

SwitchPwr <= '0';
memInt <= memInt;
cmdAck <= '0';
mainState <= cmdCapDataSelMem;
WHEN CdumpData =>
  memIntPwrl <= memIntPwrl;
  SwitchPwr <= '0';
  memInt <= memInt;
  cmdAck <= '0';
  mainState <= cmdDumpDataSelMem;
WHEN CforceOn =>
  memIntPwrl(8) <= '1'; -- Mem Int in buff will be switched on
  memIntPwrl(7 DOWNTO 0) <= buff(7 DOWNTO 0);
  SwitchPwr <= '1'; -- Signal power switch on/off
  memInt <= memInt;
  cmdAck <= '1';
  mainState <= rdCommand;
WHEN CforceOff =>
  memIntPwrl(8) <= '0'; -- Mem Int in buff will be switched off
  memIntPwrl(7 DOWNTO 0) <= buff(7 DOWNTO 0);
  SwitchPwr <= '1'; -- Signal power switch on/off
  memInt <= memInt;
  cmdAck <= '1';
  mainState <= rdCommand;
WHEN OTHERS =>
  memIntPwrl <= memIntPwrl;
  SwitchPwr <= '0';
  memInt <= memInt;
  cmdAck <= '0';
  mainState <= rdCommand;
END CASE;
ELSE
  memIntPwrl <= memIntPwrl;
  SwitchPwr <= '0';
  memInt <= memInt;
  cmdAck <= '0';
  mainState <= RdCommand;
END IF;
dataOutl <= CiOBUSDontCare;
iOBUSl <= CiOBUSDontCare;
extBUSl <= "XXXXXXXX";
hiZ <= '0';
sleepWake <= '0';
rdWrEn <= '0';
extRdWrAck <= '0';
EnableSerTPar <= '0';
enableParTSerl <= '0';
WHEN cmdSetAddrSelMem =>
  dataOutl <= CiOBUSDontCare;
  iOBUSl(31 DOWNTO 13) <= "XXXXXXXXXXXXXXXXXXXX";
  iOBUSl(12 DOWNTO 8) <= CsetAddr; -- Set GMI mode to set address
  iOBUSl(7 DOWNTO 0) <= memInt; -- Set GMI ID on LSB
  extBUSl <= "XXXXXXXX";
  memIntPwrl <= memIntPwrl;
  SwitchPwr <= '0';
  memInt <= memInt;
  hiZ <= '1';
  sleepWake <= '1';
  rdWrEn <= '0';
  cmdAck <= '0';
  extRdWrAck <= '0';
  EnableSerTPar <= '0';
  enableParTSerl <= '0';
  mainState <= cmdSetAddr;
WHEN cmdSetAddr =>
  dataOutl <= CiOBUSDontCare;
  -- write offset address to iOBUS
  iOBUSl(31 DOWNTO CmemIntWidth) <= CiOBUSDontCare(31 DOWNTO CmemIntWidth);
  iOBUSl((CmemIntWidth - 1) DOWNTO 0) <= buff((CmemIntWidth - 1) DOWNTO 0);
  extBUSl <= "XXXXXXXX";
  memIntPwrl <= memIntPwrl;
  SwitchPwr <= '0';
  memInt <= memInt;
  hiZ <= '1';
  sleepWake <= '0';

```



```

rdWrEn <= '0';
cmdAck <= '1'; -- ack command register that command has been executed
extRdWrAck <= '0';
EnableSerTPar <= '0';
enableParTSerl <= '0';
mainState <= rdCommand;
WHEN cmdWrDataSelMem =>
  dataOutl <= CiOBUSDontCare;
  iOBUSl(31 DOWNT0 13) <= "XXXXXXXXXXXXXXXXXXXX";
  iOBUSl(12 DOWNT0 8) <= CwrData; -- Set GMI mode to write 8 bit data
  iOBUSl(7 DOWNT0 0) <= memInt;
  extBUSl <= "XXXXXXXX";
  memIntPwrl <= memIntPwrl;
  SwitchPwr <= '0';
  memInt <= memInt;
  hiZ <= '1';
  sleepWake <= '1';
  rdWrEn <= '0';
  cmdAck <= '1';
  extRdWrAck <= '0';
  EnableSerTPar <= '0';
  enableParTSerl <= '0';
  mainState <= cmdWrData;
WHEN cmdWrData =>
  IF extRdWrRdy = '1' THEN -- If external RdWr event set RdWr signal to GMI's
    rdWrEn <= '1';
    extRdWrAck <= '1';
  ELSE
    rdWrEn <= '0';
    extRdWrAck <= '0';
  END IF;
  IF cmdRdy = '1' THEN -- If a new command has arrived service it
    mainState <= rdCommand;
  ELSE
    mainState <= cmdWrData; -- Otherwise keep writing data to memory
  END IF;
  dataOutl <= CiOBUSDontCare;
  iOBUSl(31 DOWNT0 8) <= "XXXXXXXXXXXXXXXXXXXX";
  iOBUSl(7 DOWNT0 0) <= ExtBus; -- Data to write will be on the External Bus
  extBUSl <= "XXXXXXXX";
  memIntPwrl <= memIntPwrl;
  SwitchPwr <= '0';
  memInt <= memInt;
  hiZ <= '1';
  sleepWake <= '0';
  cmdAck <= '0';
  EnableSerTPar <= '0';
  enableParTSerl <= '0';
WHEN cmdRdDataSelMem =>
  dataOutl <= CiOBUSDontCare;
  iOBUSl(31 DOWNT0 13) <= "XXXXXXXXXXXXXXXXXXXX";
  iOBUSl(12 DOWNT0 8) <= CrdData; -- Set GMI mode to read 8 bit data
  iOBUSl(7 DOWNT0 0) <= memInt;
  extBUSl <= "XXXXXXXX";
  memIntPwrl <= memIntPwrl;
  SwitchPwr <= '0';
  memInt <= memInt;
  hiZ <= '1';
  sleepWake <= '1';
  rdWrEn <= '0';
  cmdAck <= '0';
  extRdWrAck <= '0';
  EnableSerTPar <= '0';
  enableParTSerl <= '0';
  mainState <= cmdRdDataRelBus;
WHEN cmdRdDataRelBus => -- Tri-state IOBus and wait for one clock cycle
  dataOutl <= CiOBUSDontCare;
  iOBUSl <= CiOBUSDontCare;
  extBUSl <= "XXXXXXXX";
  memIntPwrl <= memIntPwrl;
  SwitchPwr <= '0';
  memInt <= memInt;
  hiZ <= '0'; -- Tri-state Bus
  sleepWake <= '0';
  rdWrEn <= '0';

```

```

cmdAck    <= '1';
extRdWrAck <= '0';
EnableSerTPar <= '0';
enableParTSerl <= '0';
mainState <= cmdRdData;
WHEN cmdRdData =>
  IF extRdWrRdy = '1' THEN
    rdWrEn    <= '1';
    extRdWrAck <= '1';
  ELSE
    rdWrEn    <= '0';
    extRdWrAck <= '0';
  END IF;
  IF extHiZForce = '1' THEN
    mainState <= rdCommand;
  ELSE
    mainState <= cmdRdData;
  END IF;
  dataOutl <= CiOBUSDontCare;
  iOBUSl    <= CiOBUSDontCare;
  extBUSl    <= iOBUS(7 DOWNT0 0);
  memIntPwrl <= memIntPwrl;
  SwitchPwr <= '0';
  memInt    <= memInt;
  hiZ       <= '0';
  sleepWake <= '0';
  cmdAck    <= '0';
  EnableSerTPar <= '0';
  enableParTSerl <= '0';
WHEN cmdCapDataSelMem =>
  dataOutl <= CiOBUSDontCare;
  iOBUSl(31 DOWNT0 13) <= "XXXXXXXXXXXXXXXXXXXX";
  iOBUSl(12 DOWNT0 8) <= CcapData; -- Set GMI mode to write 32 bit data
  iOBUSl(7 DOWNT0 0) <= memInt;
  extBUSl    <= "XXXXXXXX";
  memIntPwrl <= memIntPwrl;
  SwitchPwr <= '0';
  memInt    <= memInt;
  hiZ       <= '1';
  sleepWake <= '1';
  rdWrEn    <= '0';
  cmdAck    <= '1';
  extRdWrAck <= '0';
  EnableSerTPar <= '0';
  enableParTSerl <= '0';
  mainState <= cmdCapData;
WHEN cmdCapData =>
  IF cmdRdy = '1' OR stopCap = '1' THEN
    mainState <= rdCommand;
  ELSE
    mainState <= cmdCapData;
  END IF;
  dataOutl <= CiOBUSDontCare;
  iOBUSl    <= DataIn;
  extBUSl    <= "XXXXXXXX";
  memIntPwrl <= memIntPwrl;
  SwitchPwr <= '0';
  memInt    <= memInt;
  hiZ       <= '1';
  sleepWake <= '0';
  rdWrEn    <= BuffEmpty;
  cmdAck    <= '0';
  extRdWrAck <= '0';
  EnableSerTPar <= '1';
  enableParTSerl <= '0';
WHEN cmdDumpDataSelMem =>
  dataOutl <= CiOBUSDontCare;
  iOBUSl(31 DOWNT0 13) <= "XXXXXXXXXXXXXXXXXXXX";
  iOBUSl(12 DOWNT0 8) <= CDumpData; -- Set GMI mode to read 32 bit data
  iOBUSl(7 DOWNT0 0) <= memInt;

```

-- Ack command register that command has been executed

-- A command write event occurred, this is not a command,
 -- it only serves to get the data router state machine
 -- out of read mode. The next command write event will
 -- be a command again. The normal method of just writing
 -- a new command will not work, because the ExtBus is
 -- being driven by the data router
 -- Otherwise keep writing data to memory

-- If a new command is ready, or the set amount of data
 -- has been captured stop capture and return to rdCommand

-- BuffEmpty is 0 if the circbuff is empty, thus 1
 -- indicates write

[illegible]

```

-- Command Register
-- This process monitor the command write line. This line drives the command register state machine
ChangeCmdState:
PROCESS (CmdWr)
BEGIN
    IF Reset = '1' THEN
        cmdState <= rdCmd;
    ELSIF CmdWr'EVENT AND CmdWr = '1' THEN
        IF mainState = cmdRdData THEN
            cmdState <= cmdState;
            command <= command;
            buff <= buff;
            -- If the router state machine is reading data from
            -- a GMI the ExtBus will be driven by the router, thus
            -- a new command can not be recieved. This case will
            -- be reached if a dummy write was performed
        ELSE
            cmdState <= nextCmdState;
            -- Command Register state machine output latches
            command <= command;
            buff(7 DOWNTO 0) <= bufferOne;
            buff(15 DOWNTO 8) <= bufferTwo;
            buff(23 DOWNTO 16) <= bufferThree;
            buff(31 DOWNTO 24) <= bufferFour;
            buff(39 DOWNTO 32) <= bufferFive;
        END IF;
    END IF;
END PROCESS ChangeCmdState;

-- The state machine reads the first ExtBus value as a command, and then depending on what the command was the
following
-- bytes are read into a buffer up to a maximum of five bytes.
CmdStateMachine:
PROCESS (cmdState)
BEGIN
    -- The signals are defined for all states so that no unnecessary memory is created to remeber a signals previous
    state.
    CASE cmdState IS
        WHEN rdCmd =>
            command <= ExtBus;
            bufferOne <= buff(7 DOWNTO 0);
            bufferTwo <= buff(15 DOWNTO 8);
            bufferThree <= buff(23 DOWNTO 16);
            bufferFour <= buff(31 DOWNTO 24);
            bufferFive <= buff(39 DOWNTO 32);
            IF ExtBus(7 DOWNTO 5) = "000" THEN
                cmdRdy <= '1';
                nextCmdState <= rdCmd;
                -- Checks the upper three bits for number of bytes
                -- to follow first command and when the last byte is
                -- recieved command ready is signaled
            ELSE
                cmdRdy <= '0';
                nextCmdState <= rdBufferOne;
            END IF;
        WHEN rdBufferOne =>
            command <= command;
            bufferOne <= ExtBus;
            bufferTwo <= buff(15 DOWNTO 8);
            bufferThree <= buff(23 DOWNTO 16);
            bufferFour <= buff(31 DOWNTO 24);
            bufferFive <= buff(39 DOWNTO 32);
            IF command(7 DOWNTO 5) = "001" THEN
                cmdRdy <= '1';
                nextCmdState <= rdCmd;
                -- Same as above
            ELSE
                cmdRdy <= '0';
                nextCmdState <= rdBufferTwo;
            END IF;
        WHEN rdBufferTwo =>
            command <= command;
            bufferOne <= buff(7 DOWNTO 0);
            bufferTwo <= ExtBus;
            bufferThree <= buff(23 DOWNTO 16);
            bufferFour <= buff(31 DOWNTO 24);
            bufferFive <= buff(39 DOWNTO 32);
            IF command(7 DOWNTO 5) = "010" THEN
                cmdRdy <= '1';
                nextCmdState <= rdCmd;
            ELSE
                cmdRdy <= '0';
                nextCmdState <= rdBufferThree;
            END IF;
    END CASE;
END PROCESS CmdStateMachine;

```



```

ELSE
  cmdRdy1 <= '0';
  nextCmdState <= rdBufferThree;
END IF;
WHEN rdBufferThree =>
  command1 <= command;
  bufferOne <= buff(7 DOWNT0 0);
  bufferTwo <= buff(15 DOWNT0 8);
  bufferThree <= ExtBus;
  bufferFour <= buff(31 DOWNT0 24);
  bufferFive <= buff(39 DOWNT0 32);
  IF command(7 DOWNT0 5) = "011" THEN
    cmdRdy1 <= '1';
    nextCmdState <= rdCmd;
  ELSE
    cmdRdy1 <= '0';
    nextCmdState <= rdBufferFour;
  END IF;
WHEN rdBufferFour =>
  command1 <= command;
  bufferOne <= buff(7 DOWNT0 0);
  bufferTwo <= buff(15 DOWNT0 8);
  bufferThree <= buff(23 DOWNT0 16);
  bufferFour <= ExtBus;
  bufferFive <= buff(39 DOWNT0 32);
  IF command(7 DOWNT0 5) = "100" THEN
    cmdRdy1 <= '1';
    nextCmdState <= rdCmd;
  ELSE
    cmdRdy1 <= '0';
    nextCmdState <= rdBufferFive;
  END IF;
WHEN rdBufferFive =>
  command1 <= command;
  bufferOne <= buff(7 DOWNT0 0);
  bufferTwo <= buff(15 DOWNT0 8);
  bufferThree <= buff(23 DOWNT0 16);
  bufferFour <= buff(31 DOWNT0 24);
  bufferFive <= ExtBus;
  cmdRdy1 <= '1';
  nextCmdState <= rdCmd;
WHEN OTHERS =>
  command1 <= command;
  bufferOne <= buff(7 DOWNT0 0);
  bufferTwo <= buff(15 DOWNT0 8);
  bufferThree <= buff(23 DOWNT0 16);
  bufferFour <= buff(31 DOWNT0 24);
  bufferFive <= buff(39 DOWNT0 32);
  cmdRdy1 <= '0';
  nextCmdState <= rdCmd;
END CASE;
END PROCESS CmdStateMachine;

-- This process monitors the Command Write signal for a Dummy Write. When the router state machine is in the
cmdRdData
-- state the ExtBus must first be tristated before normal operation can resume. extHiZForce reset the router state
-- machine to rdCommand. In this state the ExtBus is tristated, and command can now be read from it again.
-- The command ready signal is also latched here, and a cmdRdy signal is not sent if a dummy command write was
recieved
CmdReadySignal:
PROCESS (CmdWr)
BEGIN
  IF cmdAck = '1' THEN
    cmdRdy <= '0';
    extHiZForce <= '0';
  ELSIF CmdWr'EVENT AND CmdWr = '1' THEN
    IF mainState = cmdRdData THEN
      extHiZForce <= '1';
      cmdRdy <= cmdRdy;
    ELSE
      extHiZForce <= '0';
      cmdRdy <= cmdRdy;
    END IF;
  END IF;
END PROCESS CmdReadySignal;

```

```

-- ExtBus tristate control. Only when a read command has been set up is the ExtBus driven by the data router
-- This happens on a ExtRdWr event
ExtBusCtrl:
PROCESS (ExtRdWr)
BEGIN
  IF Reset = '1' OR extHiZForce = '1' THEN
    extHiZ <= '0';
  ELSIF ExtRdWrEVENT AND ExtRdWr = '1' THEN
    IF mainState = cmdRdData THEN
      extHiZ <= '1';
    ELSE
      extHiZ <= '0';
    END IF;
  END IF;
END PROCESS ExtBusCtrl;

-- Signals the data router that a RdWr Event has occurred.
ExtRdWrRdySignal:
PROCESS (ExtRdWr)
BEGIN
  IF extRdWrAck = '1' THEN
    extRdWrRdy <= '0';
  ELSIF ExtRdWrEVENT AND ExtRdWr = '1' THEN
    IF (mainState = cmdWrData) OR (mainState = cmdRdData) THEN
      extRdWrRdy <= '1';
    ELSE
      -- If this condition is entered a command overrun has occurred. The data router was not ready to start
      -- Reading/Writing data yet.
      extRdWrRdy <= '0';
    END IF;
  END IF;
END PROCESS ExtRdWrRdySignal;

ExtBus <= ExtBusI WHEN extHiZ = '1' ELSE
  "ZZZZZZZZ";
END a;

```

B.2 Generic Memory Interface (memInt.vhd)

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY memInt0 IS
  PORT (
    -- General I/O
    Clock          : IN STD_LOGIC;
    MemIntReset    : IN STD_LOGIC;
    OverFlowIn     : IN STD_LOGIC;
    OverFRdWrIn   : IN STD_LOGIC; -- Write Overflow = 0; Read Overflow = 1
    OverFBoRWrIn  : IN STD_LOGIC; -- Byte Operations = 0; Word Operations = 1
    OverFlowOut    : OUT STD_LOGIC;
    OverFRdWrOut  : OUT STD_LOGIC;
    OverFBoRWrOut : OUT STD_LOGIC;
    Alive          : OUT STD_LOGIC;
    -- Central Data Router I/O
    IOBus          : INOUT STD_LOGIC_VECTOR(31 DOWNTO 0);
    SleepWake      : IN STD_LOGIC;
    RdWrEN         : IN STD_LOGIC;
    -- Memory Specific Interface I/O
    DataOut        : INOUT STD_LOGIC_VECTOR(31 DOWNTO 0);
    AddrOut        : OUT STD_LOGIC_VECTOR(25 DOWNTO 0);
    Rd             : OUT STD_LOGIC;
    Wr             : OUT STD_LOGIC;
    -- Testing Outputs
    HiZOut         : OUT STD_LOGIC;
    MemIntRunning  : OUT STD_LOGIC
  );

```



```

END memInt0;

ARCHITECTURE a OF memInt0 IS

    -- Generic Memory Interface ID - This is the segment of the address
    CONSTANT CmemIntID : STD_LOGIC_VECTOR(7 downto 0) := "00000000";

    -- Define memory module size = 26 bits => 64 Meg / memory module
    -- Max = 32 bits because of bus width => 4 Gig
    CONSTANT CmemIntWidth : INTEGER := 26; -- Nr of bits in each module
    CONSTANT CaddrZeroVect : STD_LOGIC_VECTOR(32 DOWNTO 0) :=
"00000000000000000000000000000000";

    -- Command Table
    CONSTANT CwrData : STD_LOGIC_VECTOR(4 DOWNTO 0) := "00000"; -- Wr Data to Ram
    CONSTANT CrdData : STD_LOGIC_VECTOR(4 DOWNTO 0) := "00001"; -- Rd Data from Ram
    CONSTANT CsetAddr : STD_LOGIC_VECTOR(4 DOWNTO 0) := "00010"; -- Setup Address
    CONSTANT CwrCmd : STD_LOGIC_VECTOR(4 DOWNTO 0) := "00100"; -- Wr Cmd to Module
    CONSTANT CrdStatus : STD_LOGIC_VECTOR(4 DOWNTO 0) := "00101"; -- Rd Status from Module
    CONSTANT CcapData : STD_LOGIC_VECTOR(4 DOWNTO 0) := "00110"; -- Capture Image Data
    CONSTANT CdumpData : STD_LOGIC_VECTOR(4 DOWNTO 0) := "00111"; -- Dump Block of Data

    -- SIGNALS
    TYPE mainStateTypes IS (start, cmdSetAddr, cmdWrData, cmdRdData, cmdCapData, cmdDumpData);

    SIGNAL mainState : mainStateTypes;
    SIGNAL mainStatel : mainStateTypes;

    SIGNAL iObusl : STD_LOGIC_VECTOR(31 DOWNTO 0);
    SIGNAL dataOutl : STD_LOGIC_VECTOR(31 DOWNTO 0);
    SIGNAL addr : STD_LOGIC_VECTOR(CmemIntWidth DOWNTO 0);

    SIGNAL command : STD_LOGIC_VECTOR(4 DOWNTO 0);

    SIGNAL running : STD_LOGIC;
    SIGNAL runningMntr : STD_LOGIC;
    SIGNAL runningOFlw : STD_LOGIC;
    SIGNAL hiZ : STD_LOGIC;
    SIGNAL zeroAddrAck : STD_LOGIC;
    SIGNAL zeroAddr : STD_LOGIC;

BEGIN
    -- This process monitors sleepwake, and when it is 1 listens on the bus for it ID.
    GMIIDMonitor:
    PROCESS (Clock, SleepWake, MemIntReset)
    BEGIN
        IF SleepWake = '0' THEN -- When sleepwake is 0 running does not change unless
            runningMntr <= runningMntr AND NOT(MemIntReset); -- a reset occurs
        ELSIF Clock'EVENT AND Clock = '1' THEN
            IF IOBus(7 DOWNTO 0) = CmemIntID THEN
                runningMntr <= '1';
                command <= IOBus(12 DOWNTO 8); -- If ID is detected, read the command bundled with it
            ELSE
                runningMntr <= '0';
                command <= command;
            END IF;
        END IF;
    END PROCESS GMIIDMonitor;

    -- Memory interface is stopped when SleepWake asserts, because this indicates a new ID selection is in progress. If
    -- this ID is reselected it will read the new command bundled with the ID. When SleepWake is 0 running depends on
    -- whether an ID was detected, or an overflow occurred.
    running <= '0' WHEN SleepWake = '1' ELSE
        runningMntr OR runningOFlw;

    -- If an overflow has taken place the memory interface is activated
    OverflowActivate:
    PROCESS (OverflowIn, MemIntReset)
    BEGIN
        IF MemIntReset = '1' OR SleepWake = '1' THEN
            runningOFlw <= '0';
        ELSIF OverflowIn'EVENT AND OverflowIn = '1' THEN
            runningOFlw <= '1';
        END IF;
    END PROCESS OverflowActivate;

```

```

-- Also monitors for an overflow, but signals state machine to zero address if overflow occurs
OverFlowZeroAddr:
PROCESS (OverFlowIn, MemIntReset)
BEGIN
    IF MemIntReset = '1' OR zeroAddrAck = '1' THEN    -- zeroAddrAck is set when zeroAddr has been recieved
        zeroAddr <= '0';
    ELSIF OverFlowIn'EVENT AND OverFlowIn = '1' THEN
        zeroAddr <= '1';
    END IF;
END PROCESS OverFlowZeroAddr;

-- In normal operations mainStatel is assigned to mainstate, but with an overflow the correct state is forced.
mainState <= cmdWrData WHEN runningOfIw = '1' AND OverFRdWrIn = '0' AND OverFBoRWIn = '0' ELSE
    cmdRdData WHEN runningOfIw = '1' AND OverFRdWrIn = '1' AND OverFBoRWIn = '0' ELSE
    cmdCapData WHEN runningOfIw = '1' AND OverFRdWrIn = '0' AND OverFBoRWIn = '1' ELSE
    cmdDumpData WHEN runningOfIw = '1' AND OverFRdWrIn = '1' AND OverFBoRWIn = '1' ELSE
    mainStatel;

-- State machine is reset whenever the memory interface is stopped
MainStateMachine:
PROCESS (running, MemIntReset, Clock)
BEGIN
    IF running = '0' OR MemIntReset = '1' THEN
        mainStatel <= start;
    ELSIF zeroAddr = '1' THEN
        addr <= CaddrZeroVect(CmemIntWidth DOWNT0 0); -- The address width is set by CmemIntWidth, and to zero
the
        zeroAddrAck <= '1';                                -- address the correct amount of zeros are specified. Note
                                                            -- that the addr vector is one bit larger than the output
                                                            -- address vector. The most significant bit is used to
                                                            -- indicate an overflow.
    ELSIF Clock'EVENT AND Clock = '1' THEN
        zeroAddrAck <= '0';                                -- send zeroAddr Ack
        CASE mainState IS
            WHEN start =>
                CASE command IS
                    WHEN CwrData =>
                        addr <= addr;
                        mainStatel <= cmdWrData;
                    WHEN CrdData =>
                        addr <= addr;
                        mainStatel <= cmdRdData;
                    WHEN CsetAddr =>
                        addr(CmemIntWidth) <= '0'; -- Clear overflow bit (msb bit) and get address from IObus
                        addr <= IOBus((CmemIntWidth - 1) DOWNT0 0);
                        mainStatel <= cmdSetAddr;
                    WHEN CcapData =>
                        addr <= addr;
                        mainStatel <= cmdCapData;
                    WHEN CdumpData =>
                        addr <= addr;
                        mainStatel <= cmdDumpData;
                    WHEN OTHERS =>
                        addr <= addr;
                        mainStatel <= start;
                END CASE;
            WHEN cmdSetAddr =>
                addr <= addr;
                mainStatel <= cmdSetAddr;
            WHEN cmdWrData =>
                IF RdWrEN = '1' THEN -- Increment the address by 1 for 8 bit operations
                    addr <= addr + '1';
                ELSE
                    addr <= addr;
                END IF;
                mainStatel <= cmdWrData;
            WHEN cmdRdData =>
                IF RdWrEN = '1' THEN -- Increment the address by 1 for 8 bit operations
                    addr <= addr + '1';
                ELSE
                    addr <= addr;
                END IF;
                mainStatel <= cmdRdData;
            WHEN cmdCapData =>

```



```

IF RdWrEn = '1' THEN
    addr <= addr + 4;           -- Increment the address by 4 for 32 bit operations
ELSE
    addr <= addr;
END IF;
mainStatel <= cmdCapData;
WHEN cmdDumpData =>
    IF RdWrEn = '1' THEN      -- Increment the address by 4 for 32 bit operations
        addr <= addr + 4;
    ELSE
        addr <= addr;
    END IF;
    mainStatel <= cmdDumpData;
WHEN OTHERS =>
    addr <= addr;
    mainStatel <= start;
END CASE;
END IF;
END PROCESS mainStateMachine;

-- Wr signal indicates a write to the Memory Specific Interface
WITH mainState SELECT
    Wr <= RdWrEn WHEN cmdWrData,
        RdWrEn WHEN cmdCapData,
        '0'      WHEN OTHERS;

-- Rd indicates a read operation to the MSI
WITH mainState SELECT
    Rd <= RdWrEn WHEN cmdRdData,
        RdWrEn WHEN cmdDumpData,
        '0'      WHEN OTHERS;

-- Drives IOBus when performing read operations
WITH mainState SELECT
    hiZ <= '1' WHEN cmdRdData,
        '1' WHEN cmdDumpData,
        '0' WHEN OTHERS;

-- Routes data from IOBus to DataOut for write operations
WITH mainState SELECT
    dataOutl <= IOBus                                     WHEN cmdWrData,
        IOBus                                     WHEN cmdCapData,
        "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"      WHEN OTHERS;

-- Routes data from MSI to IOBus for read operations
WITH mainState SELECT
    iOBusl <= DataOut                                     WHEN cmdRdData,
        DataOut                                     WHEN cmdDumpData,
        "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"      WHEN OTHERS;

-- Set overflow read/write signal; 1 = read; 0 = write
WITH mainState SELECT
    OverFRdWrOut <= '1' WHEN cmdRdData,
        '1' WHEN cmdDumpData,
        '0' WHEN cmdWrData,
        '0' WHEN cmdCapData,
        '0' WHEN OTHERS;

-- Set overflow byte/word signal; 1 = word; 0 = byte
WITH mainState SELECT
    OverFBOwWOut <= '1' WHEN cmdCapData,
        '1' WHEN cmdDumpData,
        '0' WHEN cmdWrData,
        '0' WHEN cmdRdData,
        '0' WHEN OTHERS;

Alive <= '1'; -- Signals that this memory interface device is alive

-- Tristate buffer control
IOBus <= iOBusl WHEN hiZ = '1' ELSE
    "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
DataOut <= dataOutl WHEN hiZ = '0' ELSE
    "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";

-- Assign the correct address width to MSI address
AddrOut <= addr((CmemIntWidth - 1) DOWNT0 0);
-- Use addr MSB to indicate overflow
OverFlowOut <= addr(CmemIntWidth);

-- Testing Outputs
MemIntRunning <= running; -- Indicates memint running
HiZOut <= hiZ; -- Indicates state of tristate buffers

```

END a;

B.3 Power Control (PowerCtrl.vhd)

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY PowerCtrl IS
  PORT(
    -- General I/O
    PwrReset      : IN STD_LOGIC;
    -- Data Router I/O
    MemIntPwrIn   : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    SwitchPwr     : IN STD_LOGIC;
    OnOff         : IN STD_LOGIC;
    -- Power Sitches
    MemIntPwrOut  : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
  );
END PowerCtrl;

ARCHITECTURE a OF PowerCtrl IS
  SIGNAL memIntPwrOutI : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
  -- This process reads the current power output vector and adds or removes the requested memory interface. Note
  -- that
  -- the one is added to the memory interface ID and then the LSB of the decoder is ignored. This implies that if the
  -- zero'th ID is addressed a one will be decoded, but the decoder output is wired to the power vector so that the
  -- second LSB of the decoder output is taken as the LSB of the Power vector => if the 0'th ID is selected the first
  -- bit of the power vector will be asserted.
  PROCESS (SwitchPwr, PwrReset)
  BEGIN
    IF PwrReset = '1' THEN
      memIntPwrOutI <= "00000000";
    ELSIF SwitchPwr'EVENT AND SwitchPwr = '1' THEN
      IF OnOff = '1' THEN
        memIntPwrOutI <= memIntPwrOutI OR MemIntPwrIn;
      ELSE
        memIntPwrOutI <= memIntPwrOutI AND NOT(MemIntPwrIn);
      END IF;
    END IF;
  END PROCESS;

  MemIntPwrOut <= memIntPwrOutI;
END a;
```

B.4 Circular Buffer (CircBuff.vhd)

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY CircBuff IS
  PORT(
    -- General I/O
    Clock        : IN STD_LOGIC;
    SerDataClk   : IN STD_LOGIC;
    Reset        : IN STD_LOGIC;
    Enable       : IN STD_LOGIC;
    -- Data Router I/O
    DataOut      : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
    BuffEmpty    : OUT STD_LOGIC;
    -- Shift Register I/O
    DataIn       : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    LatchData    : IN STD_LOGIC
  );
END CircBuff;
```


ARCHITECTURE a OF CircBuff IS

-- Circular Buffer elements

SUBTYPE BuffElement IS STD_LOGIC_VECTOR(31 DOWNTO 0);

TYPE BuffArray IS ARRAY (INTEGER RANGE 0 TO 3) OF BuffElement;

SIGNAL dataOutl : STD_LOGIC_VECTOR(31 DOWNTO 0);

SIGNAL head : INTEGER RANGE 0 TO 3;

SIGNAL tail : INTEGER RANGE 0 TO 3;

SIGNAL buff : BuffArray;

BEGIN

-- When LatchData is one, and a SerDataClk event occurs the data word is stored at the current head position, and
-- the head is incremented.

PushDataIntoBuff:

PROCESS (SerDataClk, LatchData, Reset, Enable)

BEGIN

IF Reset = '1' OR Enable = '0' THEN

head <= 0;

ELSIF SerDataClk'EVENT AND SerDataClk = '1' THEN

IF LatchData = '1' THEN

buff(head) <= DataIn;

head <= head + 1;

ELSE

buff(head) <= buff(head);

head <= head;

END IF;

END IF;

END PROCESS PushDataIntoBuff;

-- At each system clock event an element is read from the buffer if the buffer contains an element

PopDataFromBuff:

PROCESS (Clock, Reset, Enable)

BEGIN

IF Reset = '1' OR Enable = '0' THEN

tail <= 0;

BuffEmpty <= '0';

ELSIF Clock'EVENT AND Clock = '1' THEN

IF tail /= head THEN

tail <= tail + 1;

BuffEmpty <= '1';

DataOutl <= buff(tail);

ELSE

tail <= tail;

BuffEmpty <= '0';

DataOutl <= DataOutl;

END IF;

END IF;

END PROCESS PopDataFromBuff;

DataOut <= DataOutl;

END a;